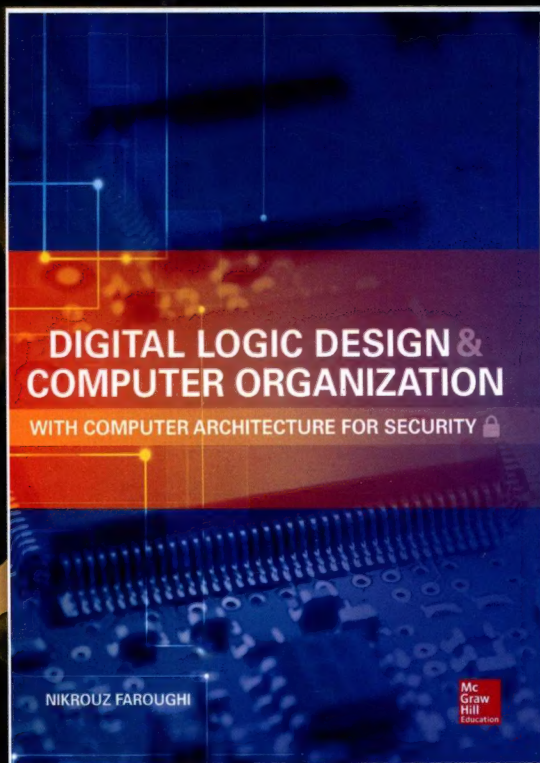


数字逻辑设计与 计算机组成

[美] 尼克罗斯·法拉菲 (Nikrouz Faroughi) 著

戴志涛 张通 黄梦凡 徐继彬 等译

Digital Logic Design and Computer Organization
with Computer Architecture for Security



计 算 机 从 书

数字逻辑设计与 计算机组成

[美] 尼克罗斯·法拉非 (Nikrouz Faroughi) 著

戴志涛 张通 黄梦凡 徐继彬 等译

Digital Logic Design and Computer Organization
with Computer Architecture for Security



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

数字逻辑设计与计算机组成 / (美) 尼克罗斯·法拉菲 (Nikrouz Faroughi) 著; 戴志涛等译. —北京: 机械工业出版社, 2017.6

(计算机科学丛书)

书名原文: Digital Logic Design and Computer Organization with Computer Architecture for Security

ISBN 978-7-111-57061-5

I. 数… II. ①尼… ②戴… III. ①数字逻辑—逻辑设计—高等学校—教材 ②计算机组成原理—高等学校—教材 IV. ① TP302.2 ② TP301

中国版本图书馆 CIP 数据核字 (2017) 第 115389 号

本书版权登记号: 图字: 01-2015-0859

Nikrouz Faroughi: Digital Logic Design and Computer Organization with Computer Architecture for Security (9780071836906)

Copyright © 2015 by McGraw-Hill Education.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2017 by McGraw-Hill Education and China Machine Press.

版权所有。未经出版人事先书面许可, 对本出版物的任何部分不得以任何方式或途径复制或传播, 包括但不限于复印、录制、录音, 或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔(亚洲)教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内(不包括香港、澳门特别行政区及台湾地区)销售。

版权 © 2017 由麦格劳-希尔(亚洲)教育出版公司与机械工业出版社所有。

本书封面贴有 McGraw-Hill Education 公司防伪标签, 无标签者不得销售。

本书从简单的数字逻辑电路设计基础开始, 由浅入深, 讲解组合逻辑电路和时序逻辑电路的设计技术、计算机组成的基本原理和计算机体系结构的相关概念, 最后深入探讨了现代计算机系统如何利用硬件支持安全的体系结构。书中通过大量实例揭示作者对现代计算机设计目标的理解, 展示如何应用流水线化和并行化技术提升并发处理能力, 并阐述了处理器体系结构与编译器、编程方法和性能之间的关系。

本书可作为高等院校“数字逻辑与计算机组成”相关课程的本科生、研究生教材, 也可作为电子信息类相关专业人士完整理解计算机系统的整体组成和硬件工作原理的参考书。

出版发行: 机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 朱秀英

责任校对: 殷虹

印刷: 中国电影出版社印刷厂

版次: 2017 年 6 月第 1 版第 1 次印刷

开本: 185mm×260mm 1/16

印张: 27.75

书号: ISBN 978-7-111-57061-5

定价: 89.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

文艺复兴以来,源远流长的科学精神和逐步形成的学术规范,使西方国家在自然科学的各个领域取得了垄断性的优势;也正是这样的优势,使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中,美国的产业界与教育界越来越紧密地结合,计算机学科中的许多泰山北斗同时身处科研和教学的最前线,由此而产生的经典科学著作,不仅擘划了研究的范畴,还揭示了学术的源变,既遵循学术规范,又自有学者个性,其价值并不会因年月的流逝而减退。

近年,在全球信息化大潮的推动下,我国的计算机产业发展迅猛,对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇,也是挑战;而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下,美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此,引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用,也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始,我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力,我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系,从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品,以“计算机科学丛书”为总称出版,供读者学习、研究及珍藏。大理石纹理的封面,也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助,国内的专家不仅提供了中肯的选题指导,还不辞劳苦地担任了翻译和审校的工作;而原书的作者也相当关注其作品在中国的传播,有的还专门为其书的中译本作序。迄今,“计算机科学丛书”已经出版了近两百个品种,这些书籍在读者中树立了良好的口碑,并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化,教育界对国外计算机教材的需求和应用都将步入一个新的阶段,我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方式如下:

华章网站: www.hzbook.com

电子邮件: hzjsj@hzbook.com

联系电话: (010) 88379604

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037



华章科技图书出版中心

信息技术发展和应用的速度如此之快，以至于层出不穷的新概念和新技术使人眼花缭乱、应接不暇。无论是物联网、移动互联网、云计算、大数据，还是人工智能、智能硬件、机器学习、智能人机交互，这些热点应用领域都要依靠计算机系统硬件提供的强大计算能力。因此，不仅是计算机专业，越来越多的各领域的专业人士都需要理解计算机系统硬件的完整组成和基本工作原理，进而在系统层面掌握计算机工作的全貌。

我在北京邮电大学从事“计算机组成原理”课程教学二十余年，深刻感觉到，计算机系统硬件课程对教和学双方而言难度都非常大。开设“计算机组成原理”课程的目的是：帮助学生理解构成计算机硬件的基本电路的特性和设计方法；使学生了解计算机系统整体概念，理解指令在计算机硬件上的执行过程；理解计算机系统层次结构，理解高级语言程序、指令系统体系结构、编译器、操作系统和硬件部件之间的关系；让学生站在系统的高度和解决问题，培养具有系统观的软件/硬件贯通人才。因此，就教师而言，如果自身没有对计算机系统硬件设计的深入体会，也就不可能让学生深入地理解计算机系统的整体组成和工作原理。

本书在计算机系统硬件教学方面做了非常有益的探索。作者 Nikrouz Faroughi 教授从密歇根州立大学获得计算机科学硕士学位、电子工程硕士学位和计算机工程方向的电子工程博士学位，并曾作为顾问和技术经理在英特尔公司工作，拥有丰富的工程经验，对计算机系统硬件有深入的了解。

本书是一部从专业角度讨论计算机系统硬件技术的完整教材。通读本书可以看出，本书与其他教科书相比具有鲜明的特色。

第一个特色是深入浅出、循序渐进，并带有丰富的教学实例。作者从简单的数字逻辑电路设计基础开始，由浅入深，讲解了组合逻辑和时序逻辑电路的设计技术、计算机组成的基本原理和计算机体系结构的相关概念。全书共 11 章。第 1 章是计算机系统相关概念的概述；第 2 ~ 5 章分别讨论组合逻辑电路和数字逻辑电路的设计；第 6 章则从大型时序电路设计过渡到计算机系统控制器的设计；第 7 ~ 9 章分别讨论存储器、指令集体系结构和系统互连（总线与输入/输出系统）；第 10 章讨论多级存储体系。

第二个特色是把计算机体系结构与系统安全问题相关联。安全问题是现代信息系统面临的重大挑战，如何在计算机系统硬件层面支持系统安全是在当今网络互连时代必须面对的问题。第 11 章用较大篇幅深入探讨了现代计算机系统如何利用硬件支持安全的体系结构。这一章从安全工程的基本概念入手，引出访问控制、硬件安全策略机制和软件/物理攻击机制，并介绍了加密技术、安全协处理器及安全通用处理器等诸多安全计算机体系结构的相关概念与方法。

第三个特色是突出现代计算机的设计理念。由于移动计算和高性能计算对系统性能和功耗的要求，以及现代半导体技术进步和信息技术发展提供的可能性，计算机组织结构的基础已经发生了改变。本书不仅提供简单的实例用于展示基本的设计概念，而且试图通过这些实例揭示作者对现代计算机设计目标的理解。书中也提供大量实例来展示如何应用流水线和

并行化技术提升并发处理能力，并揭示处理器体系结构和编译器、编程方法和性能之间的关系。

因此，本书既适合计算机及相关专业的本科学生作为相关课程的参考教科书，也可用于电子信息类相关专业人士完整理解计算机系统的整体组成和硬件工作原理。建议读者根据自身情况有选择地阅读书中的相关章节，原书前言给出了如何选择章节的建议。

本书中文版的翻译工作由我所在的北京邮电大学嵌入式系统与网络通信研究中心的教师及在校和已毕业的研究生共同完成。我与张通、黄梦凡、徐继彬、刘欣、邝坚、卞佳丽和刘健培等分别完成了初稿翻译、技术指导、术语整理和审校等工作。感谢机械工业出版社华章公司的曲熠编辑和朱秀英编辑在翻译过程中提出的诸多有益的建议。

在翻译过程中，我们一直期望在精确反映作者原意的基础上尽可能照顾到中文的表述习惯，力求在准确性和流畅性之间取得平衡。由于翻译时间仓促和译者水平有限，书中难免存在错误和疏漏，恳请读者谅解并指正。

戴志涛

2017年3月

于北京邮电大学

编写本书的目的是让读者通过一本教科书全面理解数字逻辑设计和计算机组成。此外，本书还有独立的一章介绍安全的计算机体系结构。

本书涵盖数字逻辑设计的基本原理和 Verilog 硬件描述语言设计。各个章节分别讨论简单和复杂的组合电路和时序电路的设计方法。本书概述了电路设计的现代工具和方法，而 Verilog 实例仅用于展示该语言的基本特性和可综合特性。如有需要，教师可以选择用 VHDL 替代。不过，本书并不要求使用硬件描述语言。

本书内容包括存储器组织、处理器核心和处理器组织结构，以及硬件支持的计算机安全等。由于技术的进步以及对高速和低功耗设计的需求改变了计算机组织结构的基础，因此本书尝试不仅提供简单的实例用于展示基本的设计概念，而且揭示对现代计算机设计目标的理解。

本书还从指令集体系结构角度讨论了计算机体系结构的概念，包括支持安全指令执行的架构、流水线和并行化，以及存储器层次结构。本书尝试提供大量实例来展示如何应用流水线和并行化技术来提升并发处理能力并降低或隐藏延迟（这是两个影响性能的因素）。程序代码实例也用于展示 CPU 体系结构与编译器、编程方法和性能之间的关系。

各章节概述

本书一共有 11 章。第 1 章概述了数字系统、计算创新、数码系统、数字逻辑设计和计算机组成 / 体系结构及安全。

第 2 章和第 3 章涵盖简单和复杂的组合电路，包括整型和浮点算术运算。在第 2 章讨论小型电路的设计方法时，假定若需要化简超过 4 个输入变量的真值表，学生可以利用逻辑化简软件，例如从互联网上免费下载的 Espresso。这一章还介绍了设计工具、结构级和行为级设计模型，以及利用可编程逻辑器件的电路设计，也包括 Verilog 设计实例及其综合和仿真结果。第 3 章涵盖设计大型组合电路的方法和整型及浮点数计算机算术运算，同样给出了设计实例。

第 4 ~ 6 章涵盖了简单和复杂的时序电路，从基本模型到复杂的数据通路与控制，再到时序约束、设计效率和功耗利用。第 4 章介绍了锁存器、触发器及其时序要求。第 5 章介绍了有限状态机 (FSM) 设计及其时序要求和异步输入的处理。第 6 章介绍了单周期、多周期和流水线数据通路与控制。设计实例展示了数据通路和基于 FSM、微程序及流水线的三种控制单元组织，此外还给出了几个数据通路设计实例，包括无符号和带符号乘法运算以及二维虚拟目标旋转。

第 7 章专注于存储器，包括 SDRAM 技术，以及包括交叉和多通道在内的存储器设计。这一章也介绍了存储器通信协议、性能，以及均匀存储器存取 (UMA) 和非均匀存储器存取 (NUMA) 组织；还讨论了一些编程方法实例，这些实例可以利用 NUMA 组织结构的优势来提高性能。

第 8 章讲解 CPU 设计，从单周期和流水线到精简指令集计算机 (RISC)、深度流水和分支预测，再到静态和动态指令集并行 (ILP)，直到多线程。章中包括 CPU 数据通路实例的设计和仿真，并给出了编程代码实例来说明通过编译器优化改进性能、分支预测、ILP 和多线程。

第 9 章专注于微型计算机体系结构，内容从简单的总线结构到集成结构再到现代点到点结

构的历史脉络,也包括 I/O 端口寻址、中断驱动 I/O 和直接存储器访问 (DMA),以及现代“即插即用”设备控制器接口,例如 USB 主机控制器接口。这一章还讨论了中断和相关操作系统任务,中断处理 CPU 的数据通路和指令集也被用作实例来解释简单计算机的体系结构和操作。

第 10 章涵盖存储器层次结构的原理及其组织。这一章还讨论了单处理器系统中的 cache 一致性,也介绍了共享存储器多处理器系统中的 cache 一致性问题,通过实例从缺失率、硬件数量和功耗等方面说明不同的 cache 映射技术的优势,还介绍了虚地址转换、页表管理和用于转换虚地址的可选处理器的组织结构。

第 11 章从应用在计算机体系结构中的安全工程方法的基本概念入手,接着引出访问控制、安全策略模型、硬件安全策略机制和软件/物理攻击机制,并介绍了加密技术。这一章还介绍了基于安全协处理器(用于实现安全数据存储和通信等)或者安全通用处理器实现的可信计算基(TCB)架构,也详细介绍了实现程序(指令和数据)机密性和完整性的安全处理器的体系结构。

虽然第 11 章的内容被编排在一起以方便读者阅读,但其内容也同时包含在其他章节中。例如,当学到时序电路设计技术时,学生就可以设计简单的加密电路。还有一些该章涉及的内容也在其他章节中出现过,包括硬件木马和硬件安全策略机制、存储器认证、中断的安全处理、安全协处理器和安全处理器架构等。为了给出第 11 章内容与其他章节相关内容的关联,第 1 章、第 3 章、第 5 ~ 10 章的练习部分都列出了第 11 章的练习,并加注说明为“计算机安全”。教师在这些章节中可以选择所列的习题。

为便于参考,关键字在第一次出现时会加粗显示。缩写的关键字不加粗,但少数会标出完整术语,便于读者阅读。选择本书为其课程必修教材的学术机构的教师可获得练习答案和 PowerPoint 幻灯片^①。

如何阅读本书

大多数人认为,本书对于数字设计和计算机组成的讨论深度明显高于市面上相似的教材。对于这两个主题,本教材的目标是在覆盖的广度和深度方面更加平衡。在一个学期中,教师既能审慎地选择知识点,也可以选择其课程中需要强调的每个知识点的深度和广度。本教材也包含了足够的内容来适应两个季度或两个学期的课程安排,以便深入讲授数字逻辑设计和计算机组成/体系结构两部分内容,也使学生有更多的时间来深入理解设计练习。下面是对读者阅读本书的方式的几点建议:

1. 对于没有或只有有限的数字逻辑基础的本科生,课程内容可以包括第 1 ~ 5 章和第 6 ~ 9 章的部分知识点,以及其余章节中的典型知识点。某些章节和设计实例可以跳过。
2. 对于具备一些数字逻辑基础的计算机科学和计算机工程的本科生,课程内容可以包括第 1 章、复习或选讲第 2 ~ 5 章中的部分知识点、第 6 ~ 10 章全部知识点和第 11 章的部分知识点。
3. 对于那些向没有或只有有限的数字逻辑设计和计算机组成基础的研究生授予学位的学术机构,本书是一本理想的教科书,因为其将数字逻辑设计和计算机组成及体系结构整合到了一本书中。
4. 希望更新其数字逻辑和计算机组成及体系结构知识以及希望学习安全相关的计算机体系结构概念的专业人士可以从本书中获益。

① 关于本书教辅资源,有需要的读者可向麦格劳·希尔教育出版公司北京代表处申请,电话:800 8101 936/010-6279 0299-108,电子邮件:instructorchina@mcgraw-hill.com。——编辑注

致 谢 |

Digital Logic Design and Computer Organization with Computer Architecture for Security

许多人都对本书做出了宝贵的贡献。我要特别感谢同事 Isaac Ghansah 和 Thomas Mathews, 他们对本书内容做出了重要贡献并提出了宝贵的建议。我也要感谢 Martin Nicholes (现在在英特尔工作), 他对第 11 章的内容提出了有见地的参考意见。我还要感谢审稿人给出的深思熟虑的意见, 本书终稿的部分内容根据他们的建议做了修改。

我的许多学生也对本书的初稿提供了有价值的反馈, 并帮助找出了文中的一些错误。他们对本书中某些实例的详尽分析为发现书中的文字和拼写错误帮助颇多。我要特别感谢 Kevin Schultz、Andrew Larsen、Branden Garner、Chris Dalisay、Thomas Lee、Robert Carreras、Ian Reif 和 Matt Larsen, 也欢迎大家帮助找出书中未修正的错误并提出任何改进本书的建议。我还要感谢 McGraw-Hill 出版社的赞助编辑 Michael McCabe、编辑主管 Donna Martone、著作监督 Lynn Messina、文字编辑 Lisa McCoy、美术指导 Jeff Weeks, 以及项目经理 Dheeraj Chahal 和 MPS 公司的 Surendra Shivam, 他们提供了大力支持, 为本书设计了封面并完成了最后加工。

最后但同样重要的是, 我要感谢妻子 Gita 和儿子 Kian 与 Ryon 的耐心和支持, 特别是忍受我长时间忘我地投入本书的写作。

| | | |
|-----------------|----------------|----|
| 出版者的话 | 2.5 逻辑化简算法 | 37 |
| 译者序 | 2.6 电路时序图 | 43 |
| 前言 | 2.6.1 信号传播延迟 | 45 |
| 致谢 | 2.6.2 扇入和扇出 | 45 |
| 第1章 导论 | 2.7 其他逻辑门 | 46 |
| 1.1 简介 | 2.7.1 缓存 | 46 |
| 1.1.1 数据表示 | 2.7.2 集电极开路缓冲区 | 46 |
| 1.1.2 数据通路 | 2.7.3 三态缓存 | 48 |
| 1.1.3 计算机系统 | 2.8 设计实例 | 50 |
| 1.1.4 嵌入式系统 | 2.8.1 全加器 | 50 |
| 1.2 逻辑设计 | 2.8.2 多路选择器 | 52 |
| 1.2.1 电路最小化 | 2.8.3 译码器 | 54 |
| 1.2.2 实现 | 2.8.4 编码器 | 55 |
| 1.2.3 电路类型 | 2.9 实现 | 57 |
| 1.2.4 计算机辅助设计工具 | 2.9.1 可编程逻辑器件 | 57 |
| 1.3 计算机组成 | 2.9.2 设计流程 | 58 |
| 1.4 计算机体系结构 | 2.10 硬件描述语言 | 60 |
| 1.4.1 流水线 | 2.10.1 结构模型 | 60 |
| 1.4.2 并行性 | 2.10.2 传输延迟仿真 | 63 |
| 1.5 计算机安全 | 2.10.3 行为建模 | 65 |
| 参考文献 | 2.10.4 综合与仿真 | 67 |
| 练习 | 参考文献 | 69 |
| | 练习 | 69 |
| 第2章 组合电路：小型设计 | 第3章 组合电路：大型设计 | 72 |
| 2.1 简介 | 3.1 简介 | 72 |
| 2.2 逻辑表达式 | 3.2 算术函数 | 74 |
| 2.2.1 乘积的和表达式 | 3.3 加法器 | 74 |
| 2.2.2 和的乘积表达式 | 3.3.1 进位传输加法器 | 74 |
| 2.3 规范表达式 | 3.3.2 先行进位加法器 | 75 |
| 2.3.1 极小项 | 3.4 减法器 | 81 |
| 2.3.2 极大项 | 3.5 2的补码加法/减法器 | 83 |
| 2.4 逻辑化简 | 3.6 算术逻辑单元 | 86 |
| 2.4.1 卡诺图 | 3.6.1 设计部分：位并行 | 87 |
| 2.4.2 K图化简 | 3.6.2 设计部分：位串行 | 91 |

| | | | |
|------------------------------|-----|------------------------------|-----|
| 3.7 设计实例 | 93 | 5.6.2 异步接口 | 157 |
| 3.7.1 乘法器 | 93 | 5.7 硬件描述语言模型 | 159 |
| 3.7.2 除法器 | 95 | 参考文献 | 164 |
| 3.8 实数算术 | 96 | 练习 | 164 |
| 3.8.1 浮点数标准 | 97 | | |
| 3.8.2 浮点数据空间 | 98 | 第 6 章 时序电路：大型设计 | 168 |
| 3.8.3 浮点运算 | 100 | 6.1 简介 | 168 |
| 3.8.4 浮点单元 | 104 | 6.2 数据通路设计 | 169 |
| 参考文献 | 105 | 6.2.1 单周期 | 170 |
| 练习 | 105 | 6.2.2 多周期 | 171 |
| | | 6.2.3 流水线 | 171 |
| 第 4 章 时序电路：核心模块 | 109 | 6.3 控制单元设计技术 | 175 |
| 4.1 简介 | 109 | 6.3.1 硬件控制单元：FSD | 176 |
| 4.2 SR 锁存器 | 110 | 6.3.2 微程序控制 | 176 |
| 4.3 D 锁存器 | 113 | 6.3.3 硬件控制：流水线 | 180 |
| 4.4 锁存器的缺陷 | 114 | 6.4 能源和功率消耗 | 181 |
| 4.5 D 触发器 | 115 | 6.5 设计实例 | 183 |
| 4.5.1 选择电路 | 116 | 6.5.1 无符号串行乘法器 | 184 |
| 4.5.2 操作规范 | 116 | 6.5.2 带符号串行乘法器 | 192 |
| 4.5.3 建立和保持时间 | 116 | 6.5.3 计算机图形学：旋转 | 199 |
| 4.6 无相位差的时钟频率估计 | 120 | 参考文献 | 211 |
| 4.7 触发器使能 | 120 | 练习 | 211 |
| 4.8 其他触发器 | 121 | | |
| 4.9 硬件描述语言模型 | 122 | 第 7 章 存储器 | 214 |
| 参考文献 | 124 | 7.1 简介 | 214 |
| 练习 | 125 | 7.2 存储技术 | 215 |
| | | 7.2.1 只读存储器 | 215 |
| 第 5 章 时序电路：小型设计 | 127 | 7.2.2 随机存取存储器 | 215 |
| 5.1 简介 | 127 | 7.2.3 应用 | 217 |
| 5.2 状态机介绍：寄存器设计 | 128 | 7.3 存储单元阵列 | 217 |
| 5.2.1 寄存器模型 | 129 | 7.3.1 字存取 | 218 |
| 5.2.2 多功能寄存器 | 130 | 7.3.2 突发访问 | 218 |
| 5.3 FSM 设计 | 132 | 7.4 存储器组织结构 | 220 |
| 5.3.1 二进制编码码状态 | 134 | 7.4.1 现代 DRAM | 221 |
| 5.3.2 独热码状态 | 137 | 7.4.2 SRAM 存储单元模型 | 223 |
| 5.4 计数器 | 142 | 7.4.3 SRAM 芯片内部组织结构 | 223 |
| 5.5 容错 FSM | 149 | 7.4.4 存储单元设计 | 225 |
| 5.6 时序电路的时序 | 154 | 7.5 存储时序 | 228 |
| 5.6.1 带有时钟相位差的时钟频率 | | 7.5.1 SRAM | 228 |
| 评估 | 157 | 7.5.2 DRAM | 230 |

| | | | |
|--------------------------|------------|-----------------------|------------|
| 7.5.3 SDRAM | 231 | 9.2 存储器控制器 | 298 |
| 7.5.4 DDR SDRAM | 232 | 9.2.1 简单的存储器控制器 | 298 |
| 7.6 存储器体系结构 | 232 | 9.2.2 现代存储器控制器 | 300 |
| 7.6.1 高位交叉存储 | 233 | 9.3 I/O 外围设备 | 302 |
| 7.6.2 低位交叉存储 | 233 | 9.4 控制和连接 I/O 设备 | 303 |
| 7.6.3 多通道 | 234 | 9.5 数据传输机制 | 309 |
| 7.7 设计实例: 多处理器存储结构 | 236 | 9.5.1 中断驱动传输 | 309 |
| 7.7.1 UMA 与 NUMA | 236 | 9.5.2 程序控制传输 | 311 |
| 7.7.2 NUMA 应用 | 236 | 9.5.3 DMA 传输 | 313 |
| 7.8 HDL 模型 | 237 | 9.6 中断 | 315 |
| 参考文献 | 240 | 9.6.1 中断处理 | 316 |
| 练习 | 240 | 9.6.2 中断结构 | 319 |
| 第 8 章 指令集体系结构 | 243 | 9.7 设计示例: 中断处理 CPU | 321 |
| 8.1 简介 | 243 | 9.8 USB 主控制器接口 | 325 |
| 8.1.1 指令类型 | 244 | 9.8.1 标准 | 325 |
| 8.1.2 程序翻译 | 244 | 9.8.2 事务 | 325 |
| 8.1.3 指令周期 | 244 | 9.8.3 传输 | 327 |
| 8.2 指令集体系结构的类型 | 246 | 9.8.4 描述符 | 327 |
| 8.2.1 寻址模式 | 246 | 9.8.5 帧 | 327 |
| 8.2.2 指令格式 | 247 | 9.8.6 事务组织结构 | 329 |
| 8.2.3 堆栈 ISA | 247 | 9.8.7 事务执行 | 330 |
| 8.2.4 累加器 ISA | 249 | 参考文献 | 331 |
| 8.2.5 CISC-ISA | 249 | 练习 | 331 |
| 8.2.6 RISC-ISA | 250 | 第 10 章 存储系统 | 334 |
| 8.3 设计示例 | 250 | 10.1 简介 | 334 |
| 8.3.1 累加器 ISA 指令集设计 | 250 | 10.2 cache 映射 | 338 |
| 8.3.2 累加器 ISA 处理器: 单周期 | 255 | 10.2.1 直接映射 | 339 |
| 8.3.3 累加器 ISA 处理器: 流水线 | 259 | 10.2.2 cache 缺失的类型 | 341 |
| 8.3.4 RISC-ISA 处理器 | 266 | 10.2.3 组相联映射 | 343 |
| 8.4 先进的处理器架构 | 269 | 10.3 cache 一致性 | 346 |
| 8.4.1 深度流水线 | 269 | 10.3.1 失效协议与更新协议 | 347 |
| 8.4.2 分支预测技术 | 271 | 10.3.2 监听 cache 一致性协议 | 347 |
| 8.4.3 指令级并行 | 278 | 10.3.3 直写协议 | 348 |
| 8.4.4 多线程 | 284 | 10.3.4 写回协议 | 349 |
| 参考文献 | 288 | 10.4 虚拟存储器 | 352 |
| 练习 | 288 | 10.4.1 虚拟地址转换 | 353 |
| 第 9 章 计算机体系结构: 互连 | 293 | 10.4.2 转译后备缓冲器 | 355 |
| 9.1 简介 | 293 | 10.4.3 处理器组织结构 | 356 |
| | | 参考文献 | 359 |

| | | | |
|--------------------------------|------------|-------------------------------|------------|
| 练习 | 359 | 11.8.3 应用示例：密钥链作为访问控制 | 392 |
| 第 11 章 计算机体系结构：安全 | 362 | 11.9 哈希树 | 393 |
| 11.1 简介 | 362 | 11.9.1 应用示例：密钥链认证 | 393 |
| 11.1.1 安全工程方法 | 364 | 11.9.2 应用示例：内存认证 | 393 |
| 11.1.2 威胁类型 | 365 | 11.10 安全协处理器体系结构 | 395 |
| 11.1.3 访问控制和类型 | 365 | 11.11 安全处理器体系结构 | 396 |
| 11.1.4 安全策略模型 | 367 | 11.11.1 程序代码完整性 | 396 |
| 11.1.5 攻击类型 | 369 | 11.11.2 运行安全机制 | 397 |
| 11.2 硬件后门攻击 | 370 | 11.11.3 程序代码保密性 | 399 |
| 11.2.1 数据和控制攻击 | 370 | 11.11.4 程序代码的完整性和保密性 | 399 |
| 11.2.2 定时器攻击 | 371 | 11.11.5 程序数据完整性 | 400 |
| 11.2.3 安全策略机制 | 371 | 11.11.6 程序数据保密性 | 401 |
| 11.3 软件/物理攻击 | 374 | 11.11.7 程序数据的完整性和保密性 | 404 |
| 11.3.1 欺骗攻击 | 374 | 11.11.8 程序代码和数据的完整性及保密性 | 405 |
| 11.3.2 拼接攻击 | 374 | 11.11.9 处理中断 | 406 |
| 11.3.3 重放攻击 | 375 | 11.12 设计示例：安全处理器 | 407 |
| 11.3.4 中间人攻击 | 376 | 11.12.1 SP 特征 | 407 |
| 11.4 可信计算基 | 376 | 11.12.2 处理器架构 | 408 |
| 11.5 密码使用方法 | 377 | 11.12.3 加密解密哈希引擎 | 411 |
| 11.5.1 对称密钥密码器 | 378 | 11.12.4 哈希树引擎 | 411 |
| 11.5.2 操作模式 | 379 | 11.13 延伸阅读 | 415 |
| 11.5.3 非对称密钥密码器 | 381 | 参考文献 | 417 |
| 11.6 哈希法 | 384 | 练习 | 420 |
| 11.7 加密哈希 | 386 | 参考文献 | 424 |
| 11.7.1 消息认证码 | 387 | 索引 | 426 |
| 11.7.2 基于哈希的 MAC | 387 | | |
| 11.8 通过硬件存储加密密钥 | 388 | | |
| 11.8.1 密钥链组织 | 388 | | |
| 11.8.2 存储和访问 | 388 | | |

1.1 简介

计算机、iPad、手机等设备已经引起了一场改变我们生活方方面面的数字革命。所有的数据形式，从数字和文本到音频、图像和视频，都能被表示为由一系列 0 和 1 组成的序列。数字系统已经改变了我们的沟通、工作、娱乐乃至购物的方式，并被大量应用于我们所见和所用的一事一物之上。它们也存在于汽车、杂货店结账设备、电表、机顶盒、应急设备、医用设备、工厂控制设备等系统中。随着使用数字系统的人越来越多，越来越多的数据也被创建、处理、存储、传输和访问。随之而来的是对更强大的计算机的需求，无论是个人计算机，还是用于电子商务、银行、搜索引擎和科学研究等诸多领域的大型系统。

尽管如此，计算技术的更新换代却是逐步向前推进的，其发展依赖于诸多因素，诸如集成芯片（IC）技术，以及包括操作系统在内的软件开发技术。IC 技术的发展持续推动着单个芯片中集成的晶体管数量达到数十亿个。特征尺寸，即决定作为电子开关的晶体管尺寸的 IC 元素的尺寸，多年来已变得越来越小。特征尺寸的不断缩小和裸片（die，矩形的半导体材料）尺寸的不断增长使得晶体管的密度每年大致提高 35%。这一现象又反过来促使单一芯片上的晶体管数量每 18 ~ 24 个月提高 40% ~ 55%（见文献 [1]）。这一晶体管数量提升的速率就是通常所说的摩尔定律。

近年来，微处理器设计师一直利用摩尔定律来指导未来的处理器设计。他们利用不断提升的有效晶体管数量设计出高性能的处理器，使得个人计算机发生了创造性的变革。

应用开发领域的创新也已经革新了数字系统的设计方式。当今，用于 IC 设计的计算机辅助设计（CAD）工具的发展使得芯片设计师能使用硬件描述语言（HDL）来描述数字电路的行为。这些描述可以被进一步仿真、调试、评估，甚至被自动映射到硬件上构成电路。用于电路设计的 CAD 工具已普遍应用在工业和教育领域中。

[1]

在数字领域中，也存在着对数据和信息非法访问的可能性。个人信息和众多组织的知识产权相关信息可能会被盗取、修改甚至删除。恶意软件可能会侵入私人的计算机系统或者破坏计算机操作。然而，数字系统是由硬件和软件共同构成的，并且硬件的安全性要高于软件，因此硬件对保证数字信息的安全尤为重要。

这一章简单地介绍和总结了后续章节的主要内容。本书将讨论数字系统的硬件部分，从基本电路到执行计算操作的电路模块，再到处理元素的设计——通常被称为处理核心或中央处理器（CPU）。我们也将讨论存储器、存储器系统设计，以及包含多个核心的计算机系统，即多核处理器或多处理器系统。本书也将对计算机体系结构安全做简要介绍。

1.1.1 数据表示

所有的数字系统都包括表示真或假的输入和输出逻辑的电路。逻辑值用一个电压范围表示。例如，使用 5V 电源，任何在 2.4 ~ 5V 之间的电压值表示真，任何在 0 ~ 0.8V 之间的

电压值表示假。使用电池供电的手持数字设备通常使用更低的电源以便节电。真和假的逻辑值用 1 和 0 的序列表示, 构成二进制数。

二进制数用于表示文本中的字符、图像中的像素、数字音频和视频中的数据 and 计算过程中用到的整数和实数。字符通常用 8 位美国信息交换标准码 (ASCII) 或者 16 位统一字符标准码 (Unicode) 表示。由于 ASCII 编码只能表示 $256 (2^8)$ 个不同字符 (字母、十进制数字和符号), 而与之相比, 统一字符标准编码可以表示超过 $65\,000 (2^{16})$ 个不同字符, 因此统一字符标准编码更适于表示由字组成的语言, 例如亚洲语言。

图像由显示屏上看到的数千甚至数百万个像素点构成。每一像素在彩色监视器中都由三个点值组成 (红, 绿, 蓝), 一组点值用不同的颜色或灰度组成了屏幕上的单一色点。例如, 支持真彩色的彩色监视器使用 8 位二进制数分别表示红、蓝、绿三种颜色, 那么每一种颜色就有 24 位表示, 一共可以表示超过 1600 万 (2^{24}) 种颜色。此外, 还有用 30 位或更多位表示超过十亿种颜色的色深编码方式。

数字音频和视频数据是由被数字化 (转换) 成一连串数字的模拟连续的电信号组成的。例如, 麦克风将在空气中传输的连续的声波转换成模拟电信号。然后数字转换器按固定间隔时间对该模拟电信号进行采样并生成一连串表示声音的整数值。采样间隔时间是由采样率决定的。例如, 44.1 千赫 (kHz) 的采样率意味着在一秒钟内抽取 44 100 个样本, 产生光盘 (CD) 音质的声音数据 [2]。

采样率越高, 采样的数据就越接近原始声音。每一采样值代表采样时间点的信号强度。如果用 8 位二进制数表示每一个采样值, 则信号强度被分为 256 (2^8) 级, 0 为最低级而 255 为最高级。如果用 16 位二进制数表示, 则信号强度就可以分为 65 536 (2^{16}) 级。由此看出, 用来表示信号强度的数据位数越多, 表示声波的数据就更精确, 但也需要存储更多的数据。立体声音系统由两个独立的音频声道组成。每个声道的声音都被单独采样, 这样它们所生成的声音文件大小就是单声道 (mono) 声音的两倍。尽管如此, 立体声和单声道哪个更好, 还是要取决于采样率和表示每个采样值的二进制位数的多少。

1. 整数的表示

二进制数字分为有符号和无符号两种。3 位无符号二进制数的表示范围是 $0 \sim 7$, 即二进制数 $(000)_2$ 到 $(111)_2$, 此处下标 2 表示用二进制表示。在计算机算术运算中, 有符号数通常被表示成 2 的补码。一个负的二进制数通常被各位取反 (反转) 末位加 1, 从而转换成与其等价的 2 的补码形式。例如, 用 4 位 2 的补码来表示二进制数 $-3 = -(11)_2$:

1) 将 $-(11)_2$ 写成 4 位二进制数, 即 $-(0011)_2$ 。

2) 将每一位数取反, 得 1100。

3) 对取反后的数末位加 1, 即得到 4 位 -3 的 2 的补码, 即 $(1101)_{2s}$, 其中下标 2s 表示用 2 的补码表示。

在计算机中存储的所有 2 的补码形式的数码中, 一半是正数, 另一半是负数。正数的 2 的补码形式的值与其二进制值相同。用 4 位 2 的补码形式表示 $+3 = (11)_2$ 为 $(0011)_{2s}$ 。2 的补码值的最高有效位 (MSB) 表示的是该数的符号: 如果最高有效位是 1 则该数为负数, 为 0 则为正数。将负数的 2 的补码转换成与其等价的二进制表示形式的过程与此相似, 例如将 $(1101)_{2s}$ 转换为 $-(0011)_2$ 的过程描述如下:

1) 将 $(1101)_{2s}$ 的 2 的补码表示形式各位取反, 得 0010。

2) 取反后的结果加 1, 获得 4 位二进制补码取值 $(0011)_2$ 。

3) 加上符号位, 得: $-(0011)_2$, 即 $-(11)_2 = -3$ (十进制的 -3)。

负数也可以用原码 (符号 - 数值, sm) 表示。例如, $(0011)_{sm} = +3$, 而 $(1011)_{sm} = -3$, 此处下标 sm 表示用原码表示的二进制数。在这个例子中, 最高有效位表示数的负 ($MSB = 1$) 或正 ($MSB = 0$), 其他位数表示数值的大小, 例如 $(011)_2 = 3$ 。

表 1-1 列出了与 3 位无符号数、2 的补码数和原码表示的二进制数对应的十进制数。在算术运算中, 例如加法, 可以对 2 的补码数进行运算, 原码形式仅用于来表示实数, 而不会直接使用原码形式的数进行算术运算。计算机算术运算将在第 3 章中讨论, 乘法运算会在第 6 章中讨论。

表 1-1 看作 3 位无符号数、2 的补码数和原码数时等值的十进制数

| 3 位二进制数 | 看作无符号数时等值的十进制数 | 看作 2 的补码数时等值的十进制数 | 看作原码数时等值的十进制数 |
|---------|----------------|-------------------|---------------|
| 000 | 0 | 0 | + 0 |
| 001 | 1 | 1 | 1 |
| 010 | 2 | 2 | 2 |
| 011 | 3 | 3 | 3 |
| 100 | 4 | - 4 | - 0 |
| 101 | 5 | - 3 | - 1 |
| 110 | 6 | - 2 | - 2 |
| 111 | 7 | - 1 | - 3 |

表 1-2 列出了表示 $+5$ 和 -5 的 4 位和 8 位的无符号二进制数、2 的补码形式和原码形式的表示。当 $n > m$ 时, m 位的 2 的补码数值可以转换成 n 位的 2 的补码数值, 只需简单地将符号位重复 $n - m$ 次。这一操作称为 2 的补码的符号扩展。

表 1-2 无符号数、2 的补码形式数值和原码表示方式实例

| 十进制数 | 4 位 | 8 位 | |
|------|---------------|-------------------|---|
| + 5 | $(0101)_2$ | $(00000101)_2$ | 左边补 0 扩展 |
| | $(0101)_{2s}$ | $(00000101)_{2s}$ | 应用符号扩展规则, 符号位 = 0 |
| | $(0101)_{sm}$ | $(00000101)_{sm}$ | 符号位 = 0, 左边补 0 扩展 |
| - 5 | $-(0101)_2$ | $-(00000101)_2$ | 左边补 0 扩展 |
| | $(1011)_{2s}$ | $(11111011)_{2s}$ | 应用符号扩展规则, 符号位 = 1 |
| | $(1101)_{sm}$ | $(10000101)_{sm}$ | 符号位 = 1, 左边补 0 扩展数值 $(101)_2$ 得到 7 位的数值 $(0000101)_2$ |

2. 实数的表示

计算机也使用实数进行运算, 如 2.75。实数在计算机里的表示被称作浮点 (FP) 数, 每一个浮点数都包含三个整数部分: 符号位、偏置指数和无符号尾数。符号位和无符号尾数部分组合起来就是原码表示形式。浮点算术运算包含若干步骤, 且需对指数和尾数值分别操作, 二者皆为整型数值, 本书将在第 3 章详尽讨论。

浮点数的指数是一个无符号数, 表示偏置指数。一个被称作偏置常数的固定值用来将偏移指数转换成负指数或正指数。假设一种浮点表示法使用 4 位偏移指数表示指数且偏置常数为 7。在此条件下, 偏移指数值可以为 0、15, 或者 1 ~ 14 之间的数值表示不同的浮点数。公式 (1-1) 展示了指数和与之等价的偏置指数值的关系。

$$\begin{aligned}\text{指数值} &= \text{偏置指数值} - \text{偏置常数} \\ \text{偏置指数值} &= \text{指数值} + \text{偏置常数}\end{aligned}\quad (1-1)$$

例 1-1 将实数 2.75 表示成 16 位浮点数，采用 4 位偏置指数值且取偏置常数为 7，用 11 位表示尾数。

$$\begin{aligned}2.75 &= 2 + 0.5 + 0.25 \\ &= 2 + \frac{1}{2} + \frac{1}{4} \\ &= (10)_2 + (0.1)_2 + (0.01)_2 \\ &= (10.11)_2 \\ &= (10.11)_2 \times 2^0; \text{指数} = 0 \\ &= (1.011)_2 \times 2^1; \text{指数} = 1, \text{小数点左移一位} \\ &= (1.011)_2 \times 2^8; \text{偏置指数} = 8, \text{偏置常数} = 7 \\ &=> (0, 1000, 01100000000)_2; \text{16 位浮点数表示} \\ &= 0x4300; \text{“0x” 表示十六进制数}\end{aligned}$$

16 位浮点数表示 2.75 有一位符号位 = 0 (表示正)，4 位偏置指数 = $(1000)_2$ 和 11 位无符号尾数 = $(01100000000)_2$ 。隐含的小数点在无符号尾数的最左端。尽管在 $(1.011)_2$ 中小数点前的 1 是浮点数的一部分，但它并不包含在浮点数表示法所存储在存储器中的 16 位二进制数中。类似地，-2.75 的 16 位浮点数表示为 $(1, 1000, 01100000000)_2$ ，即 0xC300，其中“0x”代表十六进制数。

假设用 k 位表示偏置指数，且偏置指数为 0，如果尾数也是 0，则该浮点数为 0.0。如果偏置指数为 0，但尾数不为 0，则该数就表示一个非常小的实数，称为非规格化数。如果偏置指数的值在 1 和 $k-2$ 之间，则该浮点数表示的值介于非常小的实数和非常大的实数之间，称为规格化浮点数。如果偏置指数为 $k-1$ ，则当尾数为 0 时，该浮点数被看作无穷大 (∞)；而当尾数不为 0 时，该浮点数被看作无效数据，例如 $\sqrt{-1}$ 。

表 1-3 4 位偏置指数和指数

| 偏置指数 | | 指 数 | | 无符号尾数 | 含 义 |
|--------|-------------|----------|----------|----------|-----------------------------------|
| 十进制数 | 二进制数 | 偏置常数 = 7 | 偏置常数 = 8 | | |
| 0 | 0000 | 0 | 0 | 0 | 表示浮点数零 (0.0) |
| 0 | 0000 | | | $\neq 0$ | 表示一个非常小的浮点数，称为非规格化数，通常不在存储器中存储 |
| 15 | 1111 | | | 0 | 表示无穷大 (例如 1.0 除以 0.0 的结果) |
| 15 | 1111 | | | $\neq 0$ | 表示一个无效的浮点数 (例如表示 $\sqrt{-1}$ 的结果) |
| 1 ~ 14 | 0001 ~ 1110 | -6 ~ 7 | -7 ~ 6 | 任何值 | 表示规格化浮点数 |

偏置常数决定了计算机中实数的表示范围。如表 1-3 所示，取 4 位偏置指数且当偏置常数 = 7 时，规格化浮点数的指数范围在 -6 ~ +7 之间；而当偏置常数 = 8 时，指数的范围在 -7 ~ +6 之间。这说明当偏置常数 = 7 时，16 位浮点数格式能表示更多的大实数：最大指数 = 7，而最小指数 = -6。而当偏置常数 = 8 时，浮点数格式能表示更多的小实数：最大指数 = 6，最小指数 = -7。现代计算机采用 32 位和 64 位电气与电子工程师协会 (IEEE) 浮

点数标准表示，这将在第 3 章中讨论。

1.1.2 数据通路

无论我们处理的数值是无符号整数、有符号整数还是浮点数，数字电路的输入和输出都是用二进制表示的。一个简单的数字电路完成一个简单函数运算并生成单比特输出。而一个复杂电路则生成多位的运算结果，完成一个或者多个函数运算。一个复杂的数字电路通常由数据通路和控制单元构成，如图 1-1 所示。图中有很多细节在这里并没有显示出来。不过，仍需要注意的是数据可以通过多条通路传输。

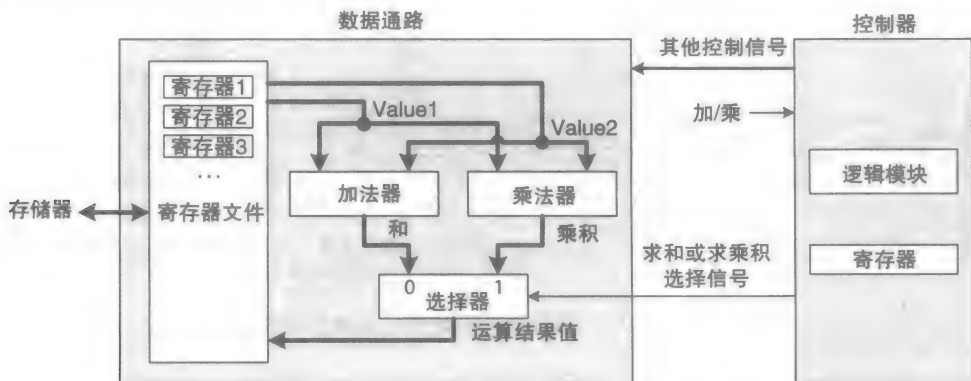


图 1-1 包含数据通路和控制器的复杂数字电路方框图

特别地，一个数据通路包含了多种电路模块，共同完成一个或者多个函数运算。而其中的模块可以是算术运算类模块，比如生成两数之和的加法器；也可以是在多个输入中择一输出的选择器；还可以是暂存数据的寄存器等。在图 1-1 中，数据通路中包含一个由多个寄存器组成的寄存器文件、一个加法器、一个乘法器和一个选择器。它可以将两个寄存器取值（图中标为 Value1 和 Value2）的和或积作为结果输出，并将结果存储在一个寄存器中。

控制器（即控制单元）产生一组信号，每一个用 1 或者 0 表示，控制数据通路的功能。例如，在图 1-1 中，当控制信号为 0 时，选择器的输出是两数之和；而当控制信号为 1 时，选择器的输出是两数之积。寄存器控制信号决定了寄存器从其输入端加载有效数值的确切时间。图中，寄存器的初始值是从存储器等外部模块中读取的，输出结果也可以存储在存储器中。

6

1.1.3 计算机系统

图 1-2 展示了被称为冯·诺依曼机的计算机系统方框图，这是迄今为止几乎所有计算机的基本架构。程序指令和数据存储在存储器中，CPU 则负责存取存储器中的指令和数据并执行指令。

CPU 是由与图 1-1 所示相似的数据通路和控制单元组成的数字电路，但是比图示要复杂得多，且包含执行三种主要操作的子数据通路：

- 取指数据通路：从存储器中加载指令。
- 译码数据通路：决定执行指令所需的控制信号。
- 执行数据通路：进行指令所要求的运算。

取指、译码和执行数据通路所完成的操作统称为指令执行。随着计算机技术的进步，

CPU 和存储器的性能近年来都不断提升,但 CPU 性能的提升速度远远高于存储器性能的提升速度。因此,冯·诺依曼体系结构在高速的 CPU 和低速的存储器之间存在通信瓶颈。

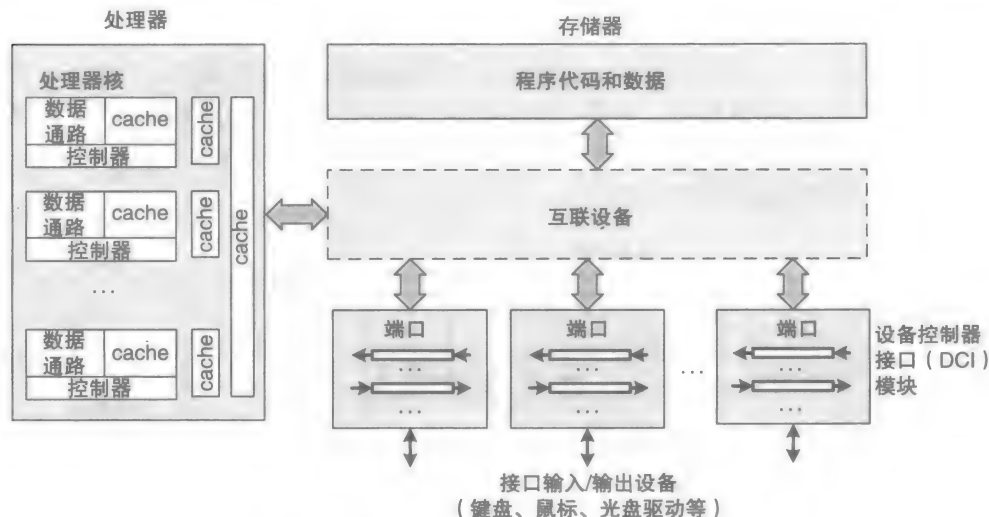


图 1-2 冯·诺依曼机的计算机系统方框图

执行数据通路可以执行一系列的特定指令,包括可以在 CPU 寄存器和存储器或者输入/输出 (I/O) 设备之间传输数据的数据传输指令。这一系列指令还包括算术运算指令,以及实现 for 循环、while 循环、子程序调用和返回等功能的指令。

编译器将高级程序语句翻译成等价的汇编指令。设有一高级程序语句“ $A = B + C;$ ”,其中 A、B 和 C 都是变量,其值存储在存储器中。采用图 1-1 所示的数据通路,则编译器将该语句翻译成等价的汇编指令,可以用不严格的语法表示如下:

```
Load R1, B    //将 B 的取值加载至寄存器 1 (R1) 中
Load R2, C    //将 C 的取值加载至寄存器 2 (R2) 中
Add R3, R1, R2 //将 R1 和 R2 的内容相加并把结果存入寄存器 3 (R3)
Store A, R3   //将求和结果存入存储器单元 A 中
```

“Load”“Add”和“Store”都是操作码,每一个操作码都有与其对应的唯一二进制数。汇编码包含数据传送指令,即“Load R1, B”和“Load R2, C”,用于将 B 的值和 C 的值从存储器中加载到寄存器 1 和寄存器 2 中。汇编码也包括算术运算指令“Add R3, R1, R2”,其中寄存器 1 和寄存器 2 为输入而寄存器 3 为输出,以及数据传输指令“Store A, R3”,用于将寄存器 3 中的值传输至存储单元 A 中。

相应的汇编程序会把每一条汇编指令翻译成二进制代码,称为机器指令。通常,汇编程序(需要时)也会链接静态库函数,比如 C 语言中的“strcpy()”“sqrt()”等例程,然后生成一个可执行(二进制)文件(例如 myprogram.exe)。程序开始执行时,首先把程序加载到存储器中,然后处理器会逐一从存储器中取出指令并译码该操作码,生成必要的控制信号执行相应指令。

被访问最频繁的指令和数据也会被存储在 cache (高速缓存) 中以提升性能。cache 存储器将从低速存储器中存取指令和数据的次数降低到最少,从而减少了从存储器中读取指令和数据的平均处理器等待时间。

设备控制器接口 (DCI) 由 I/O 端口组成, 用于处理器与键盘、光驱等外设的通信。DCI 也可以包含其他模块, 例如内部存储器, 用于在数据传输到存储器之前暂存外设数据, 或者在把数据传输至设备之前从存储器接收数据。最终, 通过互联基础架构可以将处理器、存储器、各种设备控制器接口以及用于支持与存储器通信或提升系统整体性能的其他模块互联起来。

计算机系统还可以包含特殊或专用的处理器, 例如个人计算机中的图形处理单元 (GPU) 和用在许多嵌入式系统中的数字信号处理器 (DSP)。图形处理单元和数字信号处理器都具备特殊的数据通路和控制器, 分别用于计算机图形和游戏操作及高效处理数字音频和视频数据。

1.1.4 嵌入式系统

嵌入式系统是将硬件和软件均集成在单个或多个集成电路上的完整系统。一个简单的嵌入式系统通常被称作**微控制器**, 用于设计诸如计算机键盘之类的简单设备。在单个芯片上设计的复杂嵌入式系统即片上系统 (SoC)。像手机、数字便携摄像机之类的手持设备都是嵌入式系统。嵌入式系统也用于设计先进的设备控制器接口, 比如可以和多种不同设备连接的通用串行总线 (USB) **主机控制器接口**。

嵌入式系统除了可以包含一个或者多个处理单元外, 还可根据应用的需要包含数字数据接收 / 发送模块和信号转换模块, 例如模数 (A/D) 和数模 (D/A) 转换器。模数转换器可以将模拟信号 (例如麦克风的输出信号) 转换成数字信号以便于数字通信和存储。数模转换器则相反, 比如可以将数字音频数据重新转换为模拟信号并送入扬声器。

嵌入式系统可以用定制的专用集成电路 (ASIC) 实现, 有时为了快速生成原型系统, 也采用现场可编程门阵列 (FPGA) 实现。FPGA 芯片中包含功能未确定但可以配置的电路模块。现代 FPGA 芯片包含 CPU、存储器和可配置电路模块, 可用于创建片上系统而无需进行制造实验 [3]。

9

1.2 逻辑设计

数字电路即逻辑电路, 可以实现一个或多个布尔表达式, 而每个布尔表达式均定义了一个或多个输入与单一输出之间的逻辑关系。输入和输出都由布尔变量命名, 称为信号, 每个信号的值或者为真 (T) 或者为假 (F)。公式 (1-2) 定义了有 a、b、c 三个输入信号和一个输出信号 f 的逻辑电路布尔 (逻辑) 表达式。图 1-3a 显示了该电路的方框图。

$$f = ((\text{NOT } a) \text{ AND } b) \text{ OR } c \quad (1-2)$$

表达式中的 AND、OR 和 NOT 为布尔逻辑运算符。晶体管构成的逻辑门用于实现每一种布尔运算符。现代集成电路使用数百万个逻辑门实现处理器这类复杂的逻辑电路。当两个输入都为 1 时, 与 (AND) 门的输出为 1 (真); 当至少一个输入为 1 时, 或 (OR) 门的输出为 1; 非 (NOT) 门的输出值与输入值相反, 如果输入为 0 (假) 则输出为 1, 而输入为 1 (真) 则输出为 0。例如, 当 $a = 0$ 、 $b = 1$ 且 $c = 0$ 时, f 的逻辑值由公式 (1-3) 决定:

$$\begin{aligned} f &= ((\text{NOT } 0) \text{ AND } 1) \text{ OR } 0 \\ &= (1 \text{ AND } 1) \text{ OR } 0 \\ &= 1 \text{ OR } 0 \\ &= 1 \end{aligned} \quad (1-3)$$

除了非 (NOT) 之外的其他运算符可以被扩展至多于两个布尔变量, 而其等价的逻辑门实现可支持两个或者更多个 (不超过最大值) 独立输入信号。还有其他的逻辑门, 例如与非门 (NAND) 和或非门 (NOR), 这些逻辑门可以用更少的晶体管实现。与非门在逻辑上等价于与门后面跟着一个非门 (与非逻辑), 而或非门则等价于或非逻辑。一般而言, 逻辑电路或者用与非门实现, 或者用或非门实现。1.2.2 节将讨论非门、与非门和或非门的晶体管组成。图 1-3b 显示了用非门、与门和或门 (称为与或电路) 实现表达式 f 的原理图, 图 1-3c 为与其等价的与非门电路。

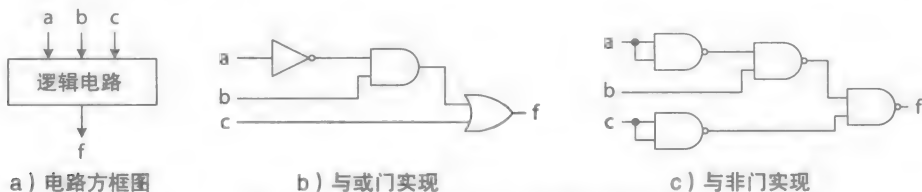


图 1-3 与或门和与非门的逻辑电路框图

如表 1-4 所示, 图 1-3 中的 a 、 b 和 c 的输入值有 8 种可能的组合。 a 、 b 和 c 可以组合起来构成一个 3 位二进制数 $(abc)_2$, 其中使 $f = 1$ 的 5 个取值 $abc = (001)_2$ 、 $(010)_2$ 、 $(011)_2$ 、 $(101)_2$ 和 $(111)_2$ 分别对应素数 1、2、3、5 和 7。剩余的令 $f = 0$ 的三个二进制数 $abc = (000)_2$ 、 $(100)_2$ 和 $(110)_2$ 则不属于素数。因此, 公式 (1-2) 定义的逻辑电路每当三位输入值 abc 代表素数时输出 1, 否则输出 0。表 1-4a 和表 1-4b 是逻辑表达式 f 的两种不同形式的真值表。随后将讨论等价表达式 g 。

表 1-4 公式 (1-2) 和公式 (1-4) 的输入逻辑组合及其对应输出值

| a) 用真或假表示逻辑值 | | | | | b) 用 1 (真) 或 0 (假) 表示逻辑值 | | | | |
|--------------|---|---|---|---|--------------------------|---|---|---|---|
| a | b | c | f | g | a | b | c | f | g |
| F | F | F | F | F | 0 | 0 | 0 | 0 | 0 |
| F | F | T | T | T | 0 | 0 | 1 | 1 | 1 |
| F | T | F | T | T | 1 | 1 | 0 | 1 | 1 |
| F | T | T | T | T | 1 | 1 | 1 | 1 | 1 |
| T | T | F | F | F | 0 | 0 | 0 | 0 | 0 |
| T | F | T | T | T | 0 | 0 | 1 | 1 | 1 |
| T | T | F | F | F | 1 | 1 | 0 | 0 | 0 |
| T | T | T | T | T | 1 | 1 | 1 | 1 | 1 |

1.2.1 电路最小化

尽管一个逻辑关系可以用若干个等价的布尔表达式表示, 但我们的目标还是要找到一个具有以下特征的最小表达式: 1) 需要更少的硬件实现 (即更少的门逻辑, 更少的门输入, 更少更短的连接线); 2) 实现该表达式的最终电路需要更短的时间产生输出结果。用布尔代数可以实现复杂表达式的最小化。

公式 (1-4) 描述了与公式 (1-2) 中 f 等价的 g 的逻辑表达式。用于实现表达式 g 的硬件部件要比实现表达式 f 的多。但是, 如表 1-4 所示, 与输出 g 和 f 相对应的每一项都是一样的。这证明公式 (1-2) 和公式 (1-4) 是等价的, 两者描述了同样的函数关系, 但公

式 (1-2) 是最小化的。实现公式 (1-2) 的电路要比实现公式 (1-4) 的电路需要少得多的硬件，而且操作速度更快。

$$\begin{aligned}
 g = & ((\text{NOT } a) \text{ AND } (\text{NOT } b) \text{ AND } c) \text{ OR} \\
 & ((\text{NOT } a) \text{ AND } b \text{ AND } (\text{NOT } c)) \text{ OR} \\
 & ((\text{NOT } a) \text{ AND } b \text{ AND } c) \text{ OR} \\
 & (a \text{ AND } (\text{NOT } b) \text{ AND } c) \text{ OR} \\
 & (a \text{ AND } b \text{ AND } c)
 \end{aligned}
 \tag{1-4}$$

11

1.2.2 实现

图 1-4 展示了使用一个 p 类和一个 n 类金属 - 氧化物半导体场效应晶体管 (MOSFET) 构成的非门电路原理图。该原理图称为 CMOS (“C” 表示互补, complement) 电路, 因为 pMOS 和 nMOS 晶体管是互补的; 当一个晶体管处于关断 (未导通) 状态时, 另一个处于开通 (导通) 状态。如图 1-4a 所示, 当输入 x 的值为逻辑 0 (0V) 时, pMOS 晶体管变为开通状态而 nMOS 晶体管变为关断状态。这将使输出信号 f 的取值如期变为逻辑 1, 正如原理图所示的指示灯点亮且电压表读数显示为 4.999V。

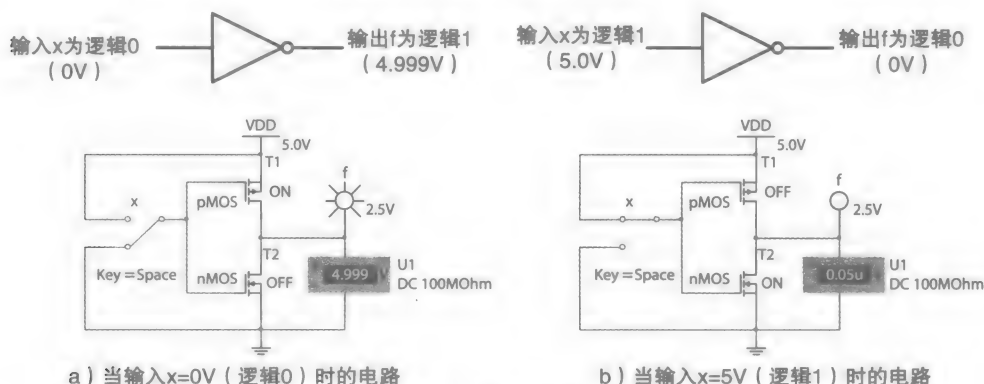


图 1-4 CMOS 非门电路原理及其电路仿真 [4]

12

在图 1-4b 中, 当输入 x 为逻辑 1 (5.0V) 时, 结果相反。pMOS 晶体管关断, 而 nMOS 晶体管开通, 输出 f 变为逻辑 0, 正如原理图所示的指示灯熄灭且电压表读数显示为 0V (0.05 μ V \approx 0V)。

老旧的大型计算机是使用低能效的门电路制造而成的, 与之相比, 现代芯片是采用高能效的 CMOS 门设计的。如图 1-4 所示, 由于其中一个晶体管始终处于关断状态, CMOS 非门基本上只在一个晶体管开通而另一个晶体管关断的情况下才消耗电能。当每次输入 x 从逻辑 1 变为逻辑 0 或者从逻辑 0 变为逻辑 1 电压时, 这种情况才会发生。当然, 如果 x 值的变换非常频繁, 晶体管在开通和关断状态之间转换得也会非常快, 这会使非门消耗更多的电能。

随着芯片上集成的晶体管的数量及其在开通和关断之间切换的速率同时提升, 芯片运转所需的电能也随之增加, 这相应地要求释放更多的热量以便冷却芯片。比如, Intel 80386 处理器功率约为 2W, 而 3.3GHz 的 Intel 酷睿 i7 处理器则消耗大约 130W 的功率 (65 倍以上)。与过去的大型计算机和超级计算机相反, 现代计算机系统采用风扇制冷系统。设计师必须考虑芯片要持续消耗多少电能, 以免超出其温度承受范围而造成芯片工作失效甚至永久性损坏。功耗问题将在第 6 章中进一步讨论。

图 1-5 显示了两输入 CMOS 与非门的电路原理图。如图所示，两个并联的 pMOS 晶体管和两个串联的 nMOS 晶体管被连接到电源和地。当两个输入 a 和 b 都为逻辑 1 时，与非门输出逻辑 0，此时两个 nMOS 晶体管均为开通状态，而两个 pMOS 晶体管均为关断状态。图中的真值表还显示了输入 a 和 b 的 4 种可能取值及其输出 f 的对应取值，以及晶体管的工作状态。

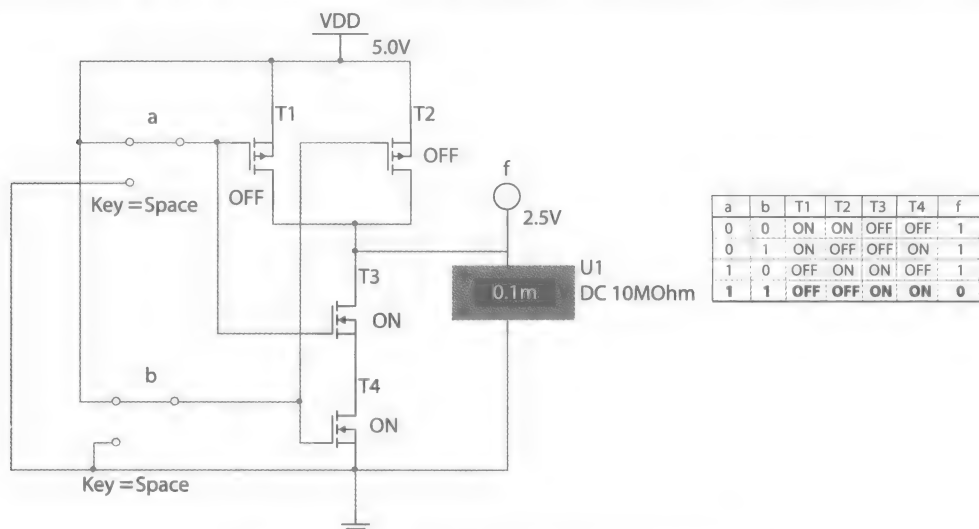


图 1-5 两个输入 CMOS 与非门的电路原理图

类似的，图 1-6 展示了两输入或非门电路原理图。在此图中，两个 pMOS 晶体管串联而两个 nMOS 晶体管并联。当至少有一个 nMOS 晶体管为开通状态时（ $a = 1$ 、 $b = 1$ 或者 $a = 1$ 且 $b = 1$ ），输出为逻辑 0（即 0）。当两个 pMOS 晶体管都为开通状态且两个 nMOS 都为关断状态时（ $a = 0$ 且 $b = 0$ ），输出为逻辑 1（即 1）。

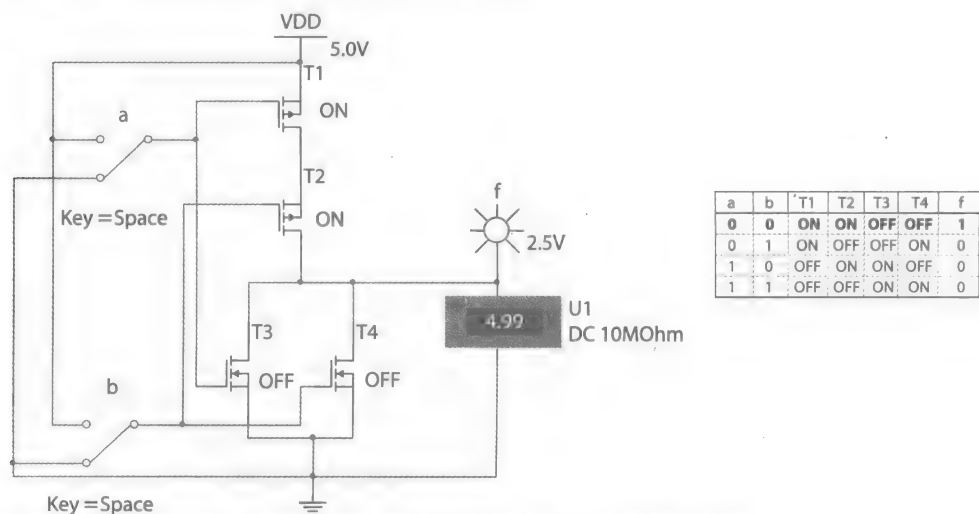


图 1-6 两个输入 CMOS 或非门的电路原理图

1.2.3 电路类型

布尔表达式既可描述组合电路，也可描述时序电路。组合电路的输出仅仅取决于电路当

前的输入值。公式 (1-2) 和公式 (1-4) 都是简单组合电路的表达式。而图 1-1 中的加法器、乘法器和数据选择器模块则是更复杂的组合电路，每个电路都产生多位输出，并基于其相应的当前输入逻辑值并行生成输出结果。

13

相反，时序电路则根据此前输入的逻辑值存储特定的状态信息。例如，输出 0、1、2、3 等值的计数器就是时序电路。它根据当前的输出值（例如 2）生成序列中的下一个取值（例如 3）。当前的计数值会作为计数器的状态存储在计数器内部。有时，时序电路也会根据一个或多个输入来决定下一个状态。例如，递增 / 递减计数器需要一个控制信号来决定计数方向。如果计数器的当前输出为 2，则递减计数时下一输出为 1，递增计数时下一输出为 3。

图 1-1 中所示的寄存器和控制器也属于时序电路。当控制器发出信号时，每个寄存器都在内部存储固定取值并一直保持该值，直到控制器再次发出信号。控制器根据一系列的状态生成控制信号去控制数据通路。一个或者多个控制信号可以用于控制数据通路中每一个模块的功能。图中，一个控制信号用于控制寄存器和数据选择器。如果逻辑值 1 用于寄存器加载而 0 用于保持，则当其控制信号为 1 时，寄存器就会从其输入端加载逻辑值，而当控制信号为 0 时，则保持其当前值不变。

假设图 1-1 中的控制器是三状态控制器，实现下述三个简单功能，并每次求出两个输入的和：

- 状态 1：从存储器加载数据至寄存器 1。
- 状态 2：从存储器加载数据至寄存器 2。
- 状态 3：选取求和结果并将其存入寄存器 3。

14

通常，一个实现诸如算术运算功能之类的电路既可以设计成组合电路，也可设计成时序电路。组合逻辑算术运算电路可以一步生成其输出结果，且并行生成所有的输出位，如图 1-7 所示。相反，时序逻辑算术运算电路用若干步串行生成最终的输出。在每一步中，时序电路都根据前一步的结果生成当前一步的输出。这个过程重复固定的次数，直到产生最终的输出结果。

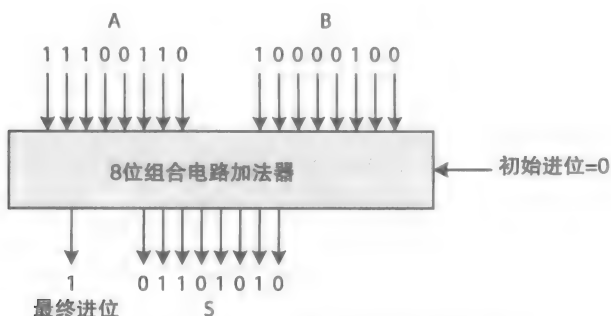


图 1-7 8 位加法器的组合电路框图

例如，一个 8 位的时序电路加法器可以重复 8 次利用单位加法器生成最终的 8 位和，如图 1-8 所示，这和我们手工进行两个数的加法运算非常相似，每次一个数字，从右往左运算。图中，在每一步运算过程中，从 A 数中取出一位，从 B 数中取出一位，再加上前一步生成的进位位生成下一步的和值。加法器内部保存进位位用于下一步操作。图中也展示了内部存储的进位位的各次运算结果。内部进位位初始值为 0。时钟信号控制数 A 和数 B 下一位进入加法器的时间以及生成和 S 的下一位的时间。经过 8 次运算，最终的 8 位和 S

就生成了。

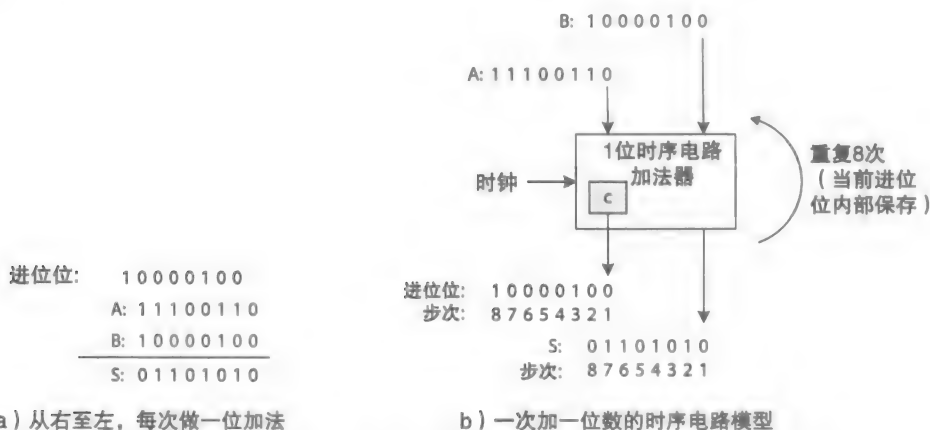


图 1-8 8 位串行加法器: a) 描述手工计算进行多位数 A 和 B 相加的过程, 每次只加一位; b) 展示了用时序电路进行多位数求和的步骤 (算法)

乘法器也可以使用组合电路或者时序电路来设计。组合电路的乘法器在同一时间运用多个组合加法模块来生成乘积。相反, 时序电路的乘法器可以重复地使用组合加法器或者时序加法器来生成乘积。相比之下, 组合运算电路通常比与其等价的时序电路速度要快, 但是也要消耗更多的硬件资源。

1.2.4 计算机辅助设计工具

布尔代数的规则也可通过软件实现, 并且已经被集成到许多逻辑设计 CAD 工具中。设计师通常使用 HDL, 或者更具体地说, 一种寄存器传送语言 (RTL) 来描述数字电路, 例如 Verilog 和 VHDL, 后者即超高速集成电路 (VHSIC) HDL。虽然可以在 RTL 中直接使用布尔表达式, 但设计师通常会使用诸如 “if-else” 语句这样的高层次描述方式来描述电路。例如, 下列语句描述了图 1-1 中数据选择模块的行为, 其中 x 用于选择加法器 (“和值”) 或者乘法器 (“乘积”) 的输出作为最终结果:

```
if (x == 0)
    result = sum;
else
    result = product;
```

以上的 “if-else” 语句与公式 (1-5) 中的逻辑表达式等价, 其中 s、p 和 r 分别表示 “和值” 的一位、“乘积” 的一位和 “结果值” 的一位:

$$r = ((\text{NOT } x) \text{ AND } s) \text{ OR } (x \text{ AND } p) \quad (1-5)$$

当 $x=0$ 时, “结果值” 即为 “和值”, 在公式 (1-5) 中取 r 的一位、 s 的一位和 p 的一位。

$$r = ((\text{NOT } 0) \text{ AND } s) \text{ OR } (0 \text{ AND } p)$$

$$r = (1 \text{ AND } s) \text{ OR } (0 \text{ AND } p)$$

$$r = s \text{ OR } 0$$

$$r = s$$

相反, 当 $x=1$ 时, r 将变为 p , 如下所示:

$$r = ((\text{NOT } 1) \text{ AND } s) \text{ OR } (1 \text{ AND } p)$$

$$r = (0 \text{ AND } s) \text{ OR } (1 \text{ AND } p)$$

$$r = 0 \text{ OR } p$$

$$r = p$$

设图 1-1 中的 sum、value1 和 value2 都是 8 位数, 则 HDL 语句 “sum = value1 + value2” 中的 + 运算符实现了类似图 1-7 中的 8 位加法器的功能, 此处进位输入和进位输出被忽略。 [16]

RTL 描述可以用于仿真, 以验证电路描述的准确性。然后, RTL 描述可以被综合 (翻译) 成被称为网表的等效最小电路表达, 用于进一步的仿真, 以验证电路的时序要求。最后, 利用 FPGA, 网表可以被自动地映射到硬件上, 生成逻辑电路。

组合和时序电路将分别在第 2 章、第 3 章和第 4 ~ 6 章详尽讨论。

1.3 计算机组成

逻辑设计要解决的是关于电路描述、综合、最小化和仿真的相关问题, 而计算机组成则研究电路部件及其物理关系, 这些部件构成处理核心 (CPU)、处理器、存储器、I/O 设备控制器和接口, 这些模块相互连接就构成计算机。例如, 图 1-1 中的寄存器文件、加法器、乘法器和选择器组成一个数据通路。控制单元和数据通路 (通过一系列控制信号) 组合成所需的运算单元, 可以产生两数之和或其乘积。两个内部组织不相同的 CPU 可以执行同一指令系统的指令。例如, 32 位 Intel 和 AMD 处理器可以执行同一指令系统的指令, 但二者的内部组织却大不相同。

计算机技术的进步也影响着计算机组成。下面列举了一些改变微型计算机 (例如, 图 1-2 所示) 组成的计算机技术的进步:

- 数据通路设计上的进步使得 CPU 得以更高效地运行, 现代处理核心 (例如 Intel 酷睿 i7) 可以并行执行多条指令。
- 存储器技术及其组织结构的进步, 例如 cache 和同步动态随机存取存储器 (SDRAM), 已经缩短了存储器的平均读/写时间, 使得处理器可以将其处理时间更多地花费在执行指令上, 而不是等待从存储器读取指令和数据。
- I/O 设备控制接口 (例如 USB 1.0、USB2.0 等) 的进步简化了个人计算机的操作。现在几乎所有的设备都支持 “即插即用”, 并且不需要设备安装和重启系统。
- 系统互联机制的进步促成了系统各组件之间更多的通信通路。采用层次化的通信通路方便更好地组织各种组件的互连。存储器与诸如处理器和 GPU 这样的高速组件之间使用高速通信通路, 而与诸如 I/O 设备这样的低速组件通信则采用低速通信通路。 [17]

不过, 供电的限制仍然制约着处理器工作速度的提高。例如, 2003 年, Intel 奔腾 4 至强处理器工作在 3.2GHz, 到差不多 7 年之后的 2010 年, Intel Nehalem 至强处理器工作在 3.33GHz, 速度仅有微小的提升 [1]。因此, 由于供电的限制, 从个人计算机到服务器, 再到超大型计算机系统 (例如仓储计算和云计算平台), 设计师构造更强大的计算机的唯一方法就是使用多处理器。

复杂电路的组织将在第 6 章讨论, 存储器设计在第 7 章讨论, CPU 设计在第 8 章讨论, 计算机设计在第 9 章讨论。

1.4 计算机体系结构

计算机组成处理的是计算机的各个部件的操作问题, 而计算机体系结构处理的是计算机

的算术运算模块（例如，加法和乘法器）和指令系统的设计，也涉及计算机系统性能提升的相关问题，包括实现在每秒钟内执行更多指令、减少程序的整体运行时间和执行更多的任务的方法。

1.4.1 流水线

流水线的概念与分级装配零件以便在更短的时间内生产出更多产品（例如汽车）的工厂装配线相似。图 1-9 的表格展示了一个简化的三级汽车装配线的工作过程。如图所示，当装配线满负荷时（每一级都有一辆汽车在组装），每 10 分钟就有一辆汽车产出（假设装配一辆车需要耗时 30 分钟：10 分钟安装发动机，10 分钟安装车门，10 分钟安装车轮）。装配线上的级数越多，且每一级的延时越短，生产的汽车就越多。例如，假设装配线可以分解成多个简单段，每段只需 2 分钟完成（即时隙为 2 分钟）。在本例中，理想的情况下，一年内可以生产出超过 260 000 辆汽车，每 2 分钟有一辆车产出。

| | | | | | | |
|---------------------|------|------|------|------|------|-----|
| 第三级：安装车轮 | | | 汽车 1 | 汽车 2 | 汽车 3 | ... |
| 第二级：安装车门 | | 汽车 1 | 汽车 2 | 汽车 3 | ... | ... |
| 第一级：安装发动机 | 汽车 1 | 汽车 2 | 汽车 3 | ... | ... | ... |
| 时隙（例如，以 10 分钟为一个时隙） | 1 | 2 | 3 | 4 | 5 | ... |

图 1-9 简化流水汽车生产线示意图

18

在设计 CPU 时，流水线概念用于将 CPU 的数据通路组织成多级，以便提高程序运行速度。考虑程序中的语句“ $A = B + C;$ ”，其等价的汇编语言程序如下：

```
Load R1, B      //将变量 B 的值从内存读入寄存器 R1 中
Load R2, C      //将变量 C 的值从内存读入寄存器 R2 中
Add R3, R1, R2  //将 R1 和 R2 的值相加并将结果存入 R3
Store A, R3     //将求和结果存入内存中的变量 A
```

图 1-10 中的表格展示了在由 3 个流水级构成的数据通路中 4 条指令的执行情况：

- 取指级：从内存中读取下一条指令。
- 译码级：生成数据通路的控制信号。
- 执行级：执行指令。

| | | | | | | |
|--------|------------|------------|----------------|----------------|----------------|-------------|
| 执行 | | | Load r1, B | Load r2, C | Add r3, r1, r2 | Store A, r3 |
| 译码 | | Load r1, B | Load r2, C | Add r3, r1, r2 | Store A, r3 | ... |
| 取指 | Load r1, B | Load r2, C | Add r3, r1, r2 | Store A, r3 | ... | ... |
| 时间 (T) | 1 | 2 | 3 | 4 | 5 | 6 |

图 1-10 流水线机制下的指令执行示意图

在时钟周期 $T = 1$ 时，从内存中取出（读出）指令“Load R1, B”； $T = 2$ 时，Load 指令正在译码级被译码，而指令“Load R2, C”被从内存中取出。 $T = 3$ 时，指令“Load R1, B”正在执行，而指令“Load R2, C”正在被译码，“Add R3, R1, R2”指令正在取指。从 $T = 3$ 开始，流水线被填满，且全部三级都处于忙状态，从时钟周期 3 到时钟周期 6，每个时钟周期都有一条指令被执行，正如图中所示。每一条指令仍然需要三个时钟周期才能执行完，但不同指令的取指、译码和执行任务可以重叠并同时完成。三级流水数据通路

由取指、译码和执行这三个独立的数据通路组成。

一般而言,虽然流水 CPU 比非流水 CPU 每秒可以执行更多的指令,但是分支和访存时间可能会延迟某些指令的执行。

浮点单元

流水线的概念也应用于复杂的算术运算模块,例如对浮点数进行操作的浮点运算单元(FPU)。浮点指令的执行需要多次算术运算和移位操作,而如果这些操作以流水方式执行,程序将会执行更快。例如,考虑以下对浮点数组进行操作的 for 循环语句:

```
float A [100], B [100], C [100];  
int i;  
for (i = 0; i < 100; i++)  
    C [i] = A [i] + B [i];
```

19

for 循环执行 100 次浮点加法指令。使用流水 CPU,这 100 条指令需要大约 100 个时钟周期完成。这减少了执行 for 循环的总时间,虽然每次执行浮点加法指令实际上都需要多个时钟周期来完成,第 3 章中将会详细讨论。在本例中,图 1-10 中所示的执行级本身将由多个流水级构成。

1.4.2 并行性

当任务相互依赖时可以采用流水线技术,例如执行指令所需的取指、译码和执行任务。另一方面,当任务相互独立且可以并行执行时则可以采用并行技术。并且,流水线技术和并行技术都要求高可用性的输入和快速的输出处理。只有当工厂装配线高效运行、必需的部件及时到达且最后成品能被快速运走时,这个工厂才能在更短的时间内生产出更多的产品。这类似于数据和指令被快速地从内存传送到处理器并且计算结果被快速存入内存。为了实现这一目标,需要用高速(cache)存储器保存最近使用的指令和数据以实现快速存取,并用低速且大容量的廉价存储器保存不同的程序和数据。

下列各小节将简要介绍应用于 CPU、处理器和系统设计的并行技术。

1. 单指令多数据流

得益于 IC 技术的发展,芯片上可用的晶体管数量不断增加,现代处理器开始包含可以在多个数据项上并行操作的特殊指令,从而提升了性能[5]。此类指令的一个例子是应用于英特尔处理器的单指令多数据流(SIMD)扩展(SSE)指令系统,或者应用在 AMD 处理器上的 3DNow 指令[6, 7]。

在许多应用领域中,例如计算机游戏,需要通过多次算术运算来生成单一的结果。一个虚拟游戏对象通常包含上千个数据点,称为顶点。在计算机屏幕上移动虚拟游戏对象需要重定位该对象的每一个顶点,采用的是一种称为顶点转换的技术。每次顶点转换都需要若干次乘法和加法运算来确定顶点的坐标位置和对象的旋转角度,使用的是将会在第 6 章中介绍的二维(2-D)坐标旋转数字计算机(CORDIC)的旋转算法。2-D CORDIC 旋转流水数据通路的设计和仿真也将在第 6 章讨论。

图 1-11 展示了包含 4 个乘法器 SIMD 数据通路,可以并行生成 4 个乘积项。在 SIMD 架构下,单个指令可以在多个数据项上进行操作,从而缩短了单个顶点转换所需的总时间。利用通常在 GPU 中常用的多 SIMD 执行单元技术,可以制作更具真实感的视频游戏。

通常,在诸如奔腾 4 之类的通用处理器中采用的 SIMD 技术仅限于支持有限的数据项,

- [20] 这对更先进的游戏来说是不够的。另一方面，GPU 则包含类似的更专用的数据通路以提升性能。

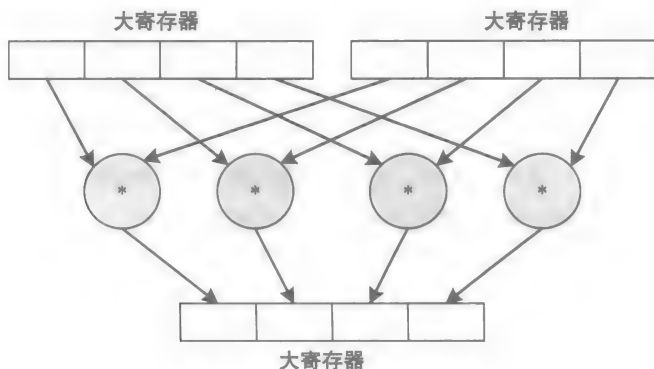


图 1-11 SIMD 多数据通路

2. 指令级并行

随着集成电路上可用的晶体管数量的增长，计算机性能的进一步提升要求并行（即同时）执行多条指令。在这种情况下，CPU 的数据通路将设计成从内存一次取出多条指令、译码多条指令并同时执行多条相互独立的指令 [8]。

确定相互独立的指令序列可以由处理器内部的硬件动态实现（例如英特尔酷睿 i7），也可以由编译器静态实现（例如英特尔基于安腾的系统）。运行于基于安腾的系统上程序所含的汇编指令由编译器组织成指令束，每个指令束最多包含三条独立指令。处理器并行地取指、译码和执行每个指令束中的指令。不过，现有的程序必须被重新编译才能利用安腾数据通路的优势，许多人相信这是其没落的原因。

图 1-12 展示了包含于一个静态生成的指令束的三条指令的指令级并行（ILP）执行过程。在每个时钟周期中，三条指令被取指、译码和执行。首先，I0 至 I2 三条指令被取指，然后当这三条指令被译码时，下三条指令 I3 至 I5 被取指。从第 3 个时钟周期开始，9 条指令被并发处理。然而，一个程序中多条指令间的数据依赖关系使得可用的硬件资源得不到最充分的利用。

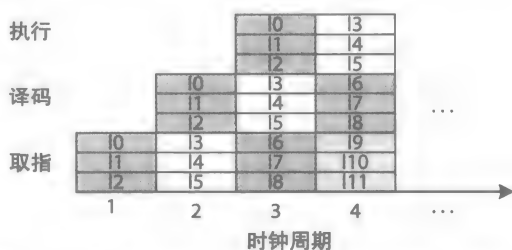


图 1-12 展示同时完成三条指令的取指、译码和执行过程的指令级并行机制

[21]

对各个基准测试程序的研究表明只有有限数量的独立指令可以同时并行执行 [9]。因此，这导致了一个限制，即到底多少个晶体管能被用于高效执行单一程序，并在执行程序时充分利用可用的硬件。因此，随着晶体管数量的不断增加，设计师并不能用过多的晶体管进一步提升处理器核的效率。有时，一个处理器核被设计为可同时运行多个（例如两个）程序以提升效率，称为多线程。更进一步，充分利用不断增长的晶体管数量的一个普遍做法是在单芯片上实现多个相同的处理核，从而构成多核处理器。ILP 和多线程体系结构将在第 8 章中讨论。

3. 多核处理器

图 1-13 展示了一个四核处理器的结构。其中的每个处理器核都可以执行一个或少量的

程序，每个程序称为一个**线程**，从而让多核处理器可以同时执行多个任务。由于在任何特定的时间点，每个处理器核都可能执行不同的指令并操作不同的数据项，因此多核处理器被认为采用了多指令多数据（MIMD）结构 [5]。回忆一下单条 SIMD 指令同时操作多个数据项的情况。同样，单指令单数据（SISD）这一术语定义了一种单核执行非 SIMD 指令的结构；然而，ILP 可以被用于加速 SISD 的执行。

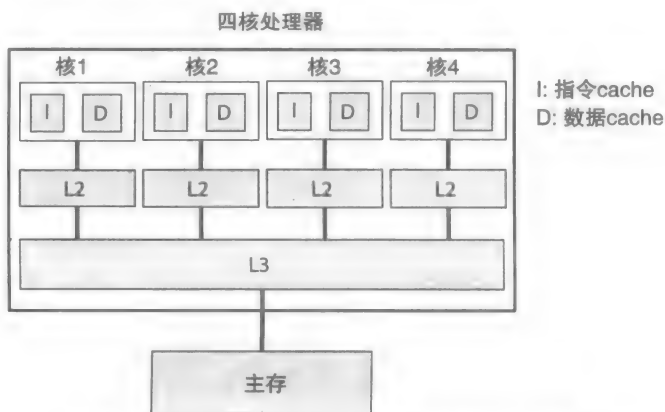


图 1-13 带三级 cache 存储器的四核处理器

22

图 1-13 同时也展示了一种共享存储器系统的结构。在此例中，为了更快地运行程序，程序员需要创建多个程序线程，这被称为**多线程编程**，方法是将程序中的数据结构在多个线程之间分割，而这些线程可以并行或并发地在多核处理器上执行，或者概括而言，在一个共享存储器的多处理器系统中运行。每个线程需要操作程序数据的子集，并与其他线程进行同步和通信。一个线程可以访问其自身的局部变量，而在多线程程序中的所有线程可以共享并操作全局声明的变量。而且，这些核可以执行不同程序的线程。

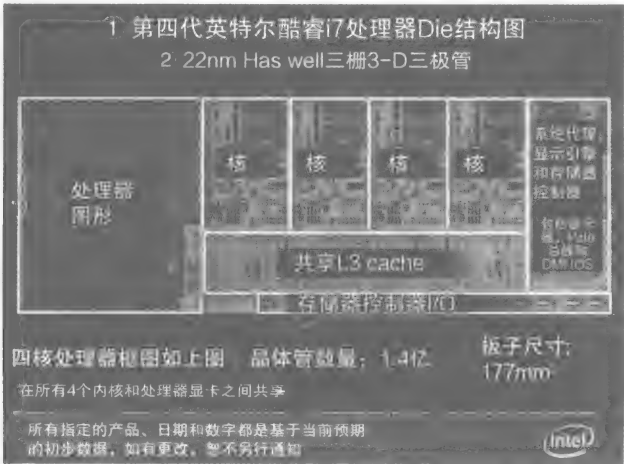
其他类型的多核处理器包含由不同复杂度的核构成的异构多核，以便支持不同应用的处理需求 [10]。例如，一个 ILP 核可以用于时序计算，而一个大型 SIMD 核可以用于并行地对大量数据项进行操作，适合也需要数据并行计算的应用。可以预见，同时需要时序和数据并行计算的应用在异构多核处理器上运行得比在同构多核处理器上要快。处理器往往使用多级 cache 存储器（如图 1-2 所示）来促进快速片上指令和数据访问，并在不同核间共享数据。

图 1-14 为第 4 代四核英特尔 i7 处理器的结构图，每个核拥有二级 cache 存储器（图中未显示）和 L3 共享 cache 存储器，以及一个图形处理器。

然而，随着片上晶体管数量及其转换速度的提高，持续的电源消耗总量和散热量也会增加。这限制了单个处理器上可以容纳的核的数量。电力和散热需求可以用称作**热设计功率**的指标度量，其中可以用于更好地平衡电源消耗和散热需求的方法有多种，例如动态降低晶体管转换速度（即时钟频率）。这些方法虽然有时可以提升处理器的性能，但是如果进一步提升性能还需要用多个处理器（每一个处理器都可以是多核）来构造多处理器系统。

4. 多处理器系统

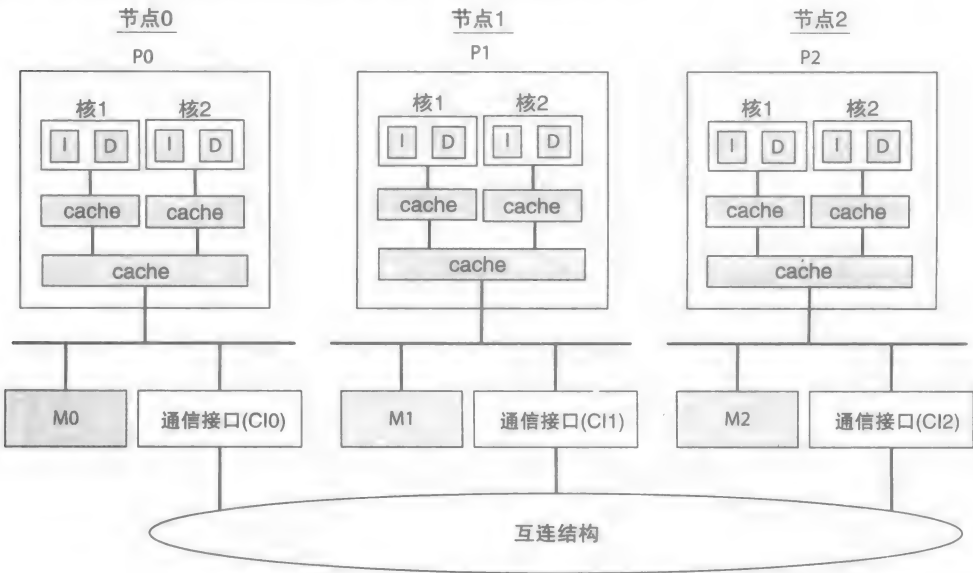
多处理器系统可以采用共享存储器机制或者**消息传递机制**。在共享存储器多处理器系统中，操作系统要有“线程感知”能力以便每个线程都能快速地访问其局部变量。然而在消息传递系统中，线程必须通过发送和接收信息来进行通信，在消息传递系统中线程没有可访问的共享变量。



23 图 1-14 第四代英特尔酷睿 i7 处理器的剖面图 (来源: 英特尔公司许可)

多处理器系统可以并行和并发地执行多个线程。因此这种工作方式提高了系统的吞吐量，即系统在单位时间内可以执行的任务数。运行一个科学应用（例如仿真海浪）所必须具备的每秒浮点运算次数（FLOPS），或者在差不多几秒钟的一小段时间内系统能够处理的谷歌搜索次数，都是系统吞吐量的实例。这些系统通常使用大量存储器并在各个单独的处理器的之间或者互联的处理器组之间分区使用。

图 1-15 展示了有三个节点的共享存储器多处理器系统结构图。在此例中，每个节点都由一个双核处理器、存储器和一个节点间通信接口组成。每个核都可以访问 M0、M1 和 M2 存储器。共享存储器多处理器系统被用于各种服务器的设计。



24 图 1-15 由三节点组成的共享存储器多处理器系统的体系结构

另一方面，多个网络系统可以构造成一个消息传递多处理器系统，其中的每一个网络系统可以是单处理器系统或者服务器节点。节点组成集群，并以通过网络发送和接收消息的方

式进行通信。最后，仓库规模计算机是由数千台服务器构成的集群。有些仓库规模的计算机被当作现代超级计算机系统用于科学计算，要求非常高的 FLOPS 指标。

集群和仓库规模计算机提供了可用性（如果一台服务器崩溃，其他的服务器还能继续工作）、交互式应用（例如在线购物、谷歌、Facebook、电子银行等）和大规模存储与计算（例如云计算）。电力分布和散热问题是当前超大型计算系统设计师面临的挑战。多核和共享存储器多处理器系统将在第 10 章中进一步讨论。

1.5 计算机安全

计算机和网络安全如今变得越来越重要，需要构造新型的计算机体系结构概念，以便能检测来自诸如病毒或间谍软件等恶意软件的攻击。个人、政府和商业组织都拥有需要防范被攻击的数字资产（程序和数据），在许多情况下也要防止员工未经授权的访问。便携设备是物理攻击的额外目标，目的是改变设备的功能或运行逆向工程任务。

因为 IC 相比于硬盘、闪存驱动器和存储器而言更加安全，所以计算机设计师们开始对一类专用和通用处理器产生了兴趣，这类处理器专门为计算机安全应用而设计并经过安全加固，例如用于实现安全的数据存储、安全通信、安全电子商务和安全程序执行等。第 11 章将介绍安全计算机体系结构。

参考文献

1. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th edition, Morgan Kaufmann, 2012.
2. Jennifer Burg, Jason Romney, and Eric Schwartz, "Digital, Sound, and Music: Concepts, Application, and Science," <http://csweb.cs.wfu.edu/~burg/CCLI/Documents/Chapter5.pdf>.
3. Xilinx FPGA, <http://www.xilinx.com/>.
4. NI Multisim, National Instruments, <http://www.ni.com/multisim/>.
5. Michael Flynn, Some computer organizations and their effectiveness, *IEEE Transactions on Computers*, Vol., No. 9, Sep. 192, pp. 948-960.
6. Intel SSE4 Programming Reference, white paper http://home.ustc.edu.cn/~shengjie/REFERENCE/sse4_instruction_set.pdf.
7. AMD 3DNow, <https://refspecs.linuxbase.org/AMD-3Dnow.pdf>.
8. Ramakrishna Rau and Fisher Joseph, Instruction-level parallel processing: History, overview, and perspective, *Journal of Supercomputing*, 7, 9-50, 1993.
9. David Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel Computer Architecture: Hardware/Software Approach*, Morgan Kaufmann, 1999.
10. Mark D. Hill, Amdahl's law in the multicore era, *IEEE Computer*, July 2008, 33-38.

练习

1.1 按要求表示下列数字：

- a. 用 4 位无符号数表示 12
- b. 用 5 位无符号数表示 12
- c. 用 4 位 2 的补码表示 +1
- d. 用 4 位 2 的补码表示 -1
- e. 用 5 位 2 的补码表示 -1
- f. 用 4 位原码表示 +1
- g. 用 4 位原码表示 -1

- 1.2 创建一个与表 1-1 类似的表格, 需要有 4 位无符号数、2 的补码数和原码三列。
- 1.3 已知浮点数有 1 位符号位, 4 位偏置指数, 11 位尾数, 其中偏置常数 = 7, 则 -5.375 的 16 位浮点数表示形式写成十六进制数是什么?
- 1.4 已知浮点数有 1 位符号位, 4 位偏置指数, 11 位尾数, 其中偏置常数 = 7, 则与浮点数 $0x3400$ 等值的实数是什么?
- 1.5 已知浮点数有 1 位符号位, 4 位偏置指数, 11 位尾数, 其中偏置常数 = 8, 则与浮点数 $0x3400$ 等值的实数是什么?
- 1.6 已知浮点数格式为 1 位符号位, 4 位偏置指数, 11 位尾数, 其中偏置常数为 7, 则可表示的 16 位最大正浮点数是什么?
- 1.7 已知浮点数格式为 1 位符号位, 4 位偏置指数, 11 位尾数, 其中偏置常数为 8, 则可表示的 16 位最大正浮点数是什么?
- 1.8 已知 16 位浮点数格式为 4 位偏置指数, 偏置常数 = 7 且 11 位尾数, 请回答:
- a. 当符号位 = 0, 偏置指数 = 1, 且尾数 = 0 时, 浮点数表示的实数是多少?
 - b. 当符号位 = 1, 偏置指数 = 14, 且尾数 = $(1111111111)_2$ 时, 浮点数表示的实数是多少?
- 1.9 用 16 位浮点数表示下列实数, 其中 4 位偏置指数, 偏置常数 = 7 且 11 位尾数:
- a. 1.0
 - b. 0.5
 - c. 0.25
- 1.10 设 16 位浮点数有 4 位偏置指数, 偏置常数 = 8 且 11 位尾数, 请回答:
- a. 当符号位 = 0, 偏置指数 = 1, 且尾数 = 0 时, 浮点数表示的实数是多少?
 - b. 当符号位 = 1, 偏置指数 = 14, 且尾数 = $(1111111111)_2$ 时, 浮点数表示的实数是多少?
- 1.11 用 16 位浮点数表示下列实数, 其中 4 位偏置指数, 偏置常数 = 8 且 11 位尾数:
- a. 1.0
 - b. 0.5
 - c. 0.25
- 1.12 画出一个与图 1-1 类似的数据通路, 用于产生高级语言程序中的语句 “ $A = A + B;$ ” 中的变量 A 的结果, 其中变量 A 和变量 B 的值在运算前从外部存储器中读入并存储在寄存器中。只能使用两个寄存器。而且, 变量 A 的初始值和 $A + B$ 的最终结果值使用同一个寄存器。请标出数据通路中所有逻辑模块并指出控制器需要完成的功能。 $A + B$ 的最终计算结果要保持在寄存器中。
- 1.13 CPU 可以执行加法、减法、乘法和除法操作。假设 4 种算术运算功能中的每一种用一个单独的模块实现, 请画出一个数据通路图, 用于产生高级语言程序中的语句 “ $A = A + B * C;$ ” 或 “ $A = A + B / C;$ ” 中的变量 A 的结果, 其中变量 A、B 和 C 的值在运算前从外部存储器中读入并存储在寄存器中。请使用不超过三个寄存器。该数据通路要能输出 $A + B * C$ 或者 $A + B / C$ 的结果。求出的最终结果要保持在寄存器中。
- 1.14 冯·诺依曼体系结构的瓶颈是什么?
- 1.15 画出三输入 CMOS 与非门的晶体管级原理图, 并给出当晶体管分别处在开和关状态时的真值表。
- 1.16 画出三输入 CMOS 或非门的晶体管级原理图, 并给出当晶体管分别处在开和关状态时的真值表。
- 1.17 CMOS 中的 “C” 代表什么? 并回答为什么这很重要?
- 1.18 流水线和并行体系结构的不同点是什么? 请说明各自的应用领域。
- 1.19 请解释晶体管数量的增长如何影响计算机的体系结构。

- 1.20 什么是高效处理核?
- 1.21 请解释为何要采用并行处理进一步提高性能。
- 1.22 画出 SIMD 数据通路以加速下列 for 循环语句的执行:

```
for(i = 0; i < 4; i++)  
    sum = sum + array[i];
```

- 1.23 画出 SIMD 数据通路以加速下列 for 循环语句的执行:

```
for(i = 0; i < 4; i++)  
    sum = sum + a[i] * b[i];
```

- 1.24 ILP 有一个限制。请问这个限制来自何处, 而处理器的设计者如何在 ILP 技术之外提升性能?
- 1.25 请解释采用多处理器系统的原因。

计算机安全

- 1.26 计算机安全 (理解安全): 选做 11.1 ~ 11.11 题。参考 11.1 节。请学生自行阅读该节。

组合电路：小型设计

2.1 简介

第1章中简单介绍了组合电路以及它们在数字系统中的应用。在这一章里，我们将涉及小的组合电路设计方法，但这些方法与设计大的组合电路用到的方法有所不同。而且，当我们手动进行最小化设计时，我们将会限制输入引脚的个数为4个，对于一些输入个数虽然超出了4但不是很大的情况（比如5个或6个），我们也会使用最小化软件进行最小化设计。对比而言，大的组合电路需要有更多的输入引脚，可以使用小的电路模块实现。关于大的组合电路设计的方法将会在第3章进行介绍。

小的组合电路中输入和输出之间的关系由真值表确定，而真值表是由描述的设计问题决定的。例如一个两位无符号数的乘法器，如图2-1中的框图所示，一个两位无符号的数 $A = a_1a_0$ 被一个两位无符号的数 $B = b_1b_0$ 乘，得到一个4位的无符号数 $P = p_3p_2p_1p_0$ 。在这个图中，大写和小写字母分别用于表示多位的和一位的输入/输出。另外，画线箭头表示了多位的输入输出（图2-1a所示）。而且，多位的输入输出也可以用粗线箭头表示（如图2-1b所示）。

表2-1表示了无符号乘法器的真值表。例如真值表中所示，由 $A = 3 = (11)_2$ 和 $B = 2 = (10)_2$ 得到的结果是 $P = 6 = (0110)_2$ 。每一个 p_3 到 p_0 的输出值都标识了一个由4位输入 a_1 、 a_0 、 b_1 和 b_0 对应的逻辑功能。

29

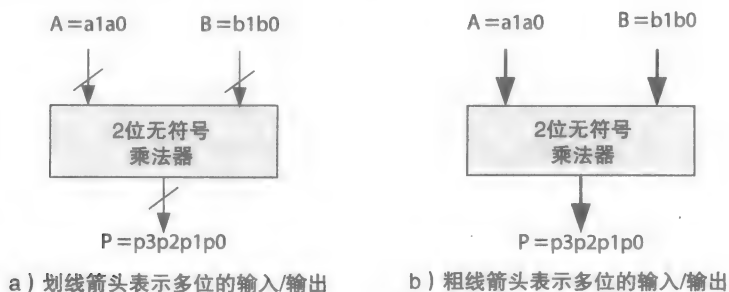


图 2-1 2 位无符号乘法器电路框图，两种方式表示多位的输入 / 输出

表 2-1 2 位无符号乘法运算模块的真值表

| 输 入 | | | | 输 出 | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|
| a_1 | a_0 | b_1 | b_0 | p_3 | p_2 | p_1 | p_0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

(续)

| 输 入 | | | | 输 出 | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|
| a_1 | a_0 | b_1 | b_0 | p_3 | p_2 | p_1 | p_0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

真值表包含了使一个输出位（比如 p_0 ）为 0 的所有的输入逻辑情况，也包含了使一个输出位为 1 的所有的输入逻辑情况。如果一个输出位始终为 0 或始终为 1，那么这个位不表示输入所对应的逻辑功能，应当在真值表中删去。真值表的行数由电路的输入引脚个数决定。如果是 3 个输入端（每个端口值为 0 或 1），就应当有 8 种可能的组合方式，或者说在真值表中应当有 8 行；如果是 4 个输入端，就应当有 16 行，如表 2-1 所示。通常情况下，如果有 n 个输入端，真值表中应当会有 2^n 行。在硬件上，真值表会有两种实现方式：

30

- 整个真值表可以存储在查找表（LUT）中，例如，表 2-1 可以存储在深度为 16 条、宽度为 4 位的存储器中。
- 一个最小的逻辑电路是由每一个与输入所对应的输出决定的。

查找表的优势在于不需要更多的设计步骤，真值表被原样存储在集成芯片内部的存储器模块中。然而，查找表的缺点却是双重的：

- 所有的 0 和 1 的输出值都要被保存，这样将使用更多的硬件资源。
- 查找表是典型的慢输出，因为它需要较长的时间去读本身的内容。

相反，一个最小的逻辑电路实现了使输出为 1 或者说使输出为 0 的输入逻辑电路。它使用更少的逻辑门、更少的输入门还有更少的连接线路，因此占用更少的硬件资源。

另一方面，存储在 LUT 中的真值表可以应用于可配置集成电路，例如现场可编程门阵列（FPGA）。FPGA 芯片中每一个查找表模块都可以用不同的真值表进行更新，去实现一个不同的组合逻辑。

在这章中的其他部分，我们将涉及如何将一个真值表转换为和它等效的逻辑表达，这种逻辑表达会在 NAND 或者 NOR 电路上实现。手工和算法的逻辑最小化技巧、最小化软件的使用、Verilog 硬件描述语言电路的描述，以及电路设计中使用的计算机辅助设计（CAD）工具，我们将举例进行讨论。这一章还提出了电路时序和潜在的时序风险。其他的门电路，比如标准门电路和三态缓冲，也都进行了讨论。这里面覆盖了很多应用，包括互连体系结构中用到的模块设计。本章中还包含了一些标准的小的组合电路模块的设计实例。

信号命名标准

回忆一下，一个信号是指电路的输入或输出为 1 或 0。同样，每一个信号名都有一个极

性指示符，该极性指示符定义了信号值 1 或者 0 在电路中的含义。信号极性定义如下：

- **高电平有效信号极性**——当信号被称为高电平有效时，逻辑 1 表示活跃、有效或使能状态；逻辑 0 代表不活跃、无效或禁止状态。通常我们用不带前缀或者后缀符号的信号名称来标识一个高电平有效信号（例如 x ）。
- **低电平有效信号极性**——当信号被称为低电平有效时，逻辑 0 表示活跃、有效或使能状态；逻辑 1 代表不活跃、无效或禁止状态。通常我们用带前缀或者后缀符号的信号名称来标识一个低电平有效信号。例如， $_x$ 、 x' 、 $/x$ 或者 $x\#$ 都可以用来表示低电平有效信号。

除非另作说明，我们一律采取下划线（ $_$ ）前缀，例如 $_x$ 或者 $_X$ 分别定义了一个低电平信号和多个低电平信号。

例 2-1 画出一位反相器的框图并在适当的地方标出其输入和输出信号。电路的输入为一位数据和标记为 $_c$ 的低电平控制信号。当 $_c$ 为不活跃（无效、禁止）时，电路的输出为与输入相比较没有改变的一位数据；当 $_c$ 为活跃（有效、使能）时，电路的输出为输入的反相。反相电路没有对输入的数据进行标记说明，方便起见，我们将输入数据和电路输出分别标记为 x 和 y 。

解：图 2-2 为有一位输入 x 、输出 y 和低电平控制信号 $_c$ 的反相电路框图。表 2-2 为一位反相器的真值表。因为 $_c$ 为低电平信号，所以当 $_c = 0$ （活跃）时， $y = \bar{x}$ ，当 $_c = 1$ （不活跃）时， $y = x$ 。

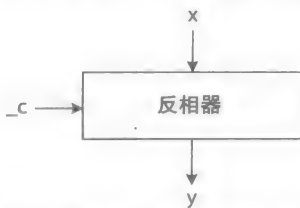


图 2-2 包含一个低电平控制信号 $_c$ 的一位反相电路框图

表 2-2 包含一个低电平控制信号 $_c$ 的一位反相器的真值表

| $_c$ | x | y |
|-------|-----|-----|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

2.2 逻辑表达式

图 2-3 展示了用于实现逻辑电路的符号和逻辑门的真值表。与（AND）门、或（OR）门、与非（NAND）门、或非（NOR）门、非（NOT）门在第 1 章中有过讨论。异或（XOR）门、同或（XNOR）门是有两个输入的逻辑门。当两个输入相同时，异或门的输出为 0，反之为 1。相反，当两个输入不同时，同或门输出为 0，反之为 1。异或门、同或门都可以看作一位比较器。与非门、或非门是通用的逻辑门，因为它们可以用来实现任何逻辑表达式。除此之外，实现它们时需要的晶体管更少。在集成电路内部，所有的逻辑实现都是用与非门、或非门实现的。

表 2-3 展示了有两个变量的函数 f ，并定义当 $x = 0$ 且 $y = 0$ 或者 $x = 1$ 且 $y = 1$ 时， f 的值为 1，当 x 和 y 取其他情况下的值时， f 的值为 0。公式（2-1）是一种表示真值表的布尔表达式，点（“ \cdot ”）表示与运算，“+”表示或运算，横线表示非运算。

$$f = \bar{x} \cdot \bar{y} + x \cdot y \tag{2-1}$$

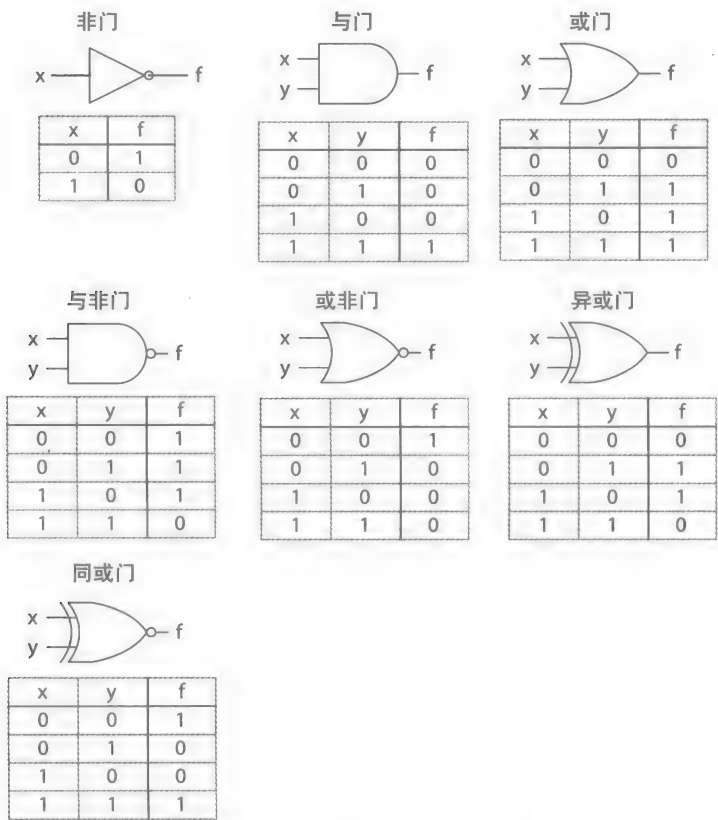


图 2-3 基础的逻辑门符号和对应的真值表

表 2-3 两变量函数

| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

33

如果取遍了一个逻辑表达式的输入值都能产生出相同的真值表，那么这个逻辑表达式就等价于其真值表。例如，公式 (2-1) 在表 2-3 的输入下产生相同的输出，如下所示：

$x = 0 \text{ 且 } y = 0:$

$f = \overline{0} \cdot \overline{0} + 0 \cdot 0 = 1 \cdot 1 + 0 \cdot 0 = 1 + 0 = 1$

$x = 0 \text{ 且 } y = 1:$

$f = \overline{0} \cdot \overline{1} + 0 \cdot 1 = 1 \cdot 0 + 0 \cdot 0 = 0 + 0 = 0$

$x = 1 \text{ 且 } y = 0:$

$f = \overline{1} \cdot \overline{0} + 0 \cdot 1 = 0 \cdot 1 + 0 \cdot 1 = 0 + 0 = 0$

$x = 1 \text{ 且 } y = 1:$

$f = \overline{1} \cdot \overline{1} + 1 \cdot 1 = 0 \cdot 0 + 1 \cdot 1 = 0 + 1 = 1$

公式 (2-1) 只包含了两个逻辑项 $\overline{x} \cdot \overline{y}$ 和 $x \cdot y$ ，它们对应使得 f 的值为 1 的输入逻辑。通常，与 (AND) 运算可以不用任何符号表示 (没有 “ \cdot ”), 如公式 (2-2) 所示。

$$f = \overline{x} \overline{y} + xy$$

(2-2)

2.2.1 乘积的和表达式

当一个布尔表达式的输入满足使其输出信号 $f = 1$ 的条件时 (例如公式 (2-2)), 该表

达式称为乘积的和 (SOP)。表达式中的每一项都用每个独立输入条件相与 (乘积) 的形式给出, 因而构成一个乘积项。当一个或者更多的乘积项为 1 时, 函数输出 f 为 1; 因此, f 就是乘积项的或 (加法) 运算的结果。在公式 (2-2) 中, 当 x 和 y 都为 0 时, $\bar{x}\bar{y} = 1$, 或者 x 和 y 都为 1 时, $xy = 1$, 因此 $f = 1$ 。

一个 SOP 表达式可以翻译成与其等价的由非门、与门和或门基本逻辑门组成的逻辑电路。与公式 (2-2) 相对应的电路框图由两级与-或结构组成, 不包含非门, 如图 2-4 所示。该电路由 7 个信号组成: 输入信号 x 和 y ; 输出信号 f ; 中间信号 $t = \bar{x}$, $u = \bar{y}$, $g = \bar{t}\bar{u}$ 和 $h = xy$ 。两个与门电路生成中间信号 g 和 h , 作为或门的输入信号用来产生最终输出 f 。

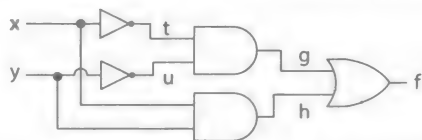


图 2-4 函数 $f = \bar{x}\bar{y} + xy$ 的与或门级结构框图

运用非门、与门和或门来实现一个逻辑表达式比用通用的与非门和或非门实现更直接。

用与非门实现 SOP 表达式

用与非门代替与-或电路中所有的非门、与门和或门可以生成一个只由与非门组成的等价电路。这种转换基于德摩根定律, 如公式 (2-3) 所示。与非运算符被表示为与门连接一个非门的形式 (例如 \overline{xy}), 或非运算符被表示为或门连接一个非门的形式 (例如 $\overline{x+y}$):

$$\text{定律 1: } \overline{xy} = \bar{x} + \bar{y}$$

$$\text{定律 2: } \overline{x+y} = \bar{x}\bar{y}$$

(2-3)

定律 1 中的 \overline{xy} 和 $\bar{x} + \bar{y}$ 两个表达式可以生成相同的真值表, 所以说它们是等价的。同样, 定律 2 中的 $\overline{x+y}$ 和 $\bar{x}\bar{y}$ 也是等价的。这些定律同样适用于若干个变量。定律 1 表明了一个有反相输入的或门和一个没有反相输入的与非门是逻辑等价的。定律 2 表明了一个有反相输入的与门和一个没有反相输入的或非门是逻辑等价的。

考虑图 2-4 中的与-或电路。电路可以通过以下步骤转换成只有与非门的电路:

1) 将一个二输入的与非门的输入连接起来, 作为与非门等价的与非门逻辑代替电路中的每一个非门, 如下图所示, 即 $y = \bar{x} \cdot \bar{x} = \bar{x}$ 。



2) 在中间信号 g 和 h 的两端加入两个非门, 如图 2-5 所示。这个操作并不会影响电路行为, 因为两个非门不会改变原始信号的值 (例如 $\bar{\bar{g}} = g$)。

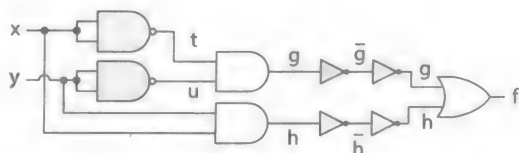


图 2-5 用两个加在线路两端的非门来实现 g 和 h 的与或电路图

3) 用与非门替换每一个与-非门的电路组合, 如图 2-6 所示; 一个与非门等价于一个与门后面跟着一个非门的组合。内部非门在图 2-5 中用小圆圈表示。

唯一一个没有转换成与非门的是有反相输入的或门。根据德摩根定律 1: $\bar{g}h = \bar{g} + \bar{h}$, 这样的逻辑门是和与非门等价的。

4) 用与非门代替有反相输入的或门的电路框图如图 2-7 所示。

图 2-7 中由与非门组成的电路结构等价于图 2-4 中的与-或电路结构。另外, 下列布尔

代数运算得出了一个只有与非运算的 SOP 表达式：

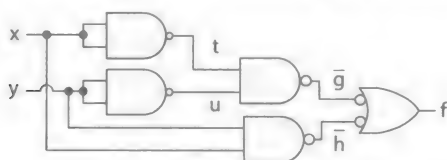


图 2-6 用内部非门实现的
与-或电路图

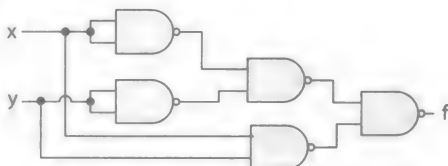


图 2-7 只用与非门实现 SOP 表
达式 $f = xy + \bar{x}\bar{y}$ 的电路图

35

回想一下， $f = \bar{\bar{f}}$ 。用 SOP 表达式 $\bar{x}\bar{y} + xy$ 代替式中的 \bar{f} 得：

$$f = \overline{\bar{x}\bar{y} + xy}$$

在式子中运用德摩根定律 2，用 $(\bar{x}\bar{y})(xy)$ 代替 $\bar{x}\bar{y} + xy$ 。两个与非项 $\bar{x}\bar{y}$ 和 xy 又组成了一项新的与非项来表达 f ，这样 f 中就只有与非项，如下所示：

$$f = (\bar{x}\bar{y})(xy)$$

2.2.2 和的乘积表达式

当一个布尔表达式定义了输出信号 f 的输入条件使得 $f = 0$ 时，我们称其为和的乘积 (POS)。上节中讨论了一个 SOP 的定义为输出信号的输入条件使得 $f = 1$ 。POS 表达式中的每一项都用每个独立输入条件相或 (加法) 的形式给出，使得 f 为 0，这样就构成了一个和项；当一个和项为 0 时，信号 f 的值为 0，这样 f 就是它所有和项的与 (乘积) 结果。同一个输出的 SOP 和 POS 表达式是等价的，它们有相同的真值表。我们只需要用一种表达式 (SOP 或者 POS) 来描述输出信号即可。然而，SOP 表达式比 POS 表达式更容易从直观上理解。

1. SOP 与 POS

下列规则展示了输出 f 的 SOP 表达式和 POS 表达式之间的关系：

- 规则 1: f 的 POS 表达式 = \bar{f} 的 SOP 表达式的补集。
- 规则 2: f 的 SOP 表达式 = \bar{f} 的 POS 表达式的补集。

36

运用规则 1，函数 f 的 POS 表达式可由 \bar{f} 的 SOP 表达式的补集得到。在真值表中，当 f 为 0 时， \bar{f} 为 1，当 f 为 1 时， \bar{f} 为 0。由于 \bar{f} 的 SOP 表达式定义了 \bar{f} 在输入条件下为 1，所以 \bar{f} 的 SOP 表达式的补集就是一个定义当输入条件下 f 为 0 的 POS 表达式，如下表所示：

| x | y | f | \bar{f} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$f \text{ 的 SOP} = \bar{x}y + x\bar{y}$$

这样，可推得：

$$f = \overline{\bar{f} \text{ 的 SOP}} = \overline{\bar{x}y + x\bar{y}}$$

由德摩根定律 2，即 $\overline{\bar{x}y} = \bar{x}\bar{y}$ ，得：

$$f = (\bar{x}\bar{y})(x\bar{y})$$

由德摩根定律 1, 即 $\overline{xy} = \bar{x} + \bar{y}$, 对于每一个逻辑项, 得:

$$f = (\bar{x} + \bar{y})(\bar{x} + \bar{y})$$

或者写为:

$$f \text{ 的 POS} = (x + \bar{y})(\bar{x} + y) \quad (2-4)$$

等式中的两个逻辑项 $(x + \bar{y})$ 和 $(\bar{x} + y)$ 都为和 (即 OR) 项, 这两个和项通过与运算组成了一个 POS 表达式。规则 2 可用于从 \bar{f} 的 POS 表达式推出 f 的 SOP 表达式。

另外, f 的 POS 表达式可以由对偶原理得出 \bar{f} 的 SOP 表达式:

对偶原理——表达式 $\bar{x}y + x\bar{y}$ 的对偶表达式为 $(\bar{x} + y)(x + \bar{y})$, 即与和或运算符号互换, 与运算都换成或运算, 或运算都换成与运算, 但是无论是对偶形式还是非对偶形式, 变量名是不变的。

通常, 一个布尔代数规则的对偶形式也是合法的布尔代数规则, 例如 $x(y + z) = xy + xz$ 的对偶形式 $x + yz = (x + y)(x + z)$ 也是合法的布尔代数规则。如果布尔代数规则里含有 1 或者 0, 在求其对偶的布尔代数规则时, 需将 0 换成 1 或者 1 换成 0, 例如 $x + 0 = x$ 就转换成 $x \cdot 1 = x$ 。

通过真值表获得 POS 表达式 $f = (x + \bar{y})(\bar{x} + y)$ 的一个简单的方法是首先获得 $\bar{f} = \bar{x}y + x\bar{y}$ 的对偶表达式, 然后将其对偶表达式每个变量取反就可获得 POS 表达式 $f = (x + \bar{y})(\bar{x} + y)$ 。

例 2-2 已知 $\bar{g} = \bar{x}yz + xy\bar{z} + x\bar{y}z$ 的 SOP 表达式, 求 g 的 POS 表达式。

解: 首先求出 \bar{g} 的对偶表达式:

$$\bar{g} \text{ 的对偶表达式} = (\bar{x} + y + z)(x + y + \bar{z})(x + \bar{y} + z)$$

将对偶表达式中的每一个变量取反, 就可得到 g 的 POS 表达式:

$$g \text{ 的 POS 表达式} = (x + \bar{y} + \bar{z})(\bar{x} + \bar{y} + z)(\bar{x} + y + \bar{z})$$

通过函数 \bar{f} 的 SOP 表达式求出 f 的 POS 表达式的两种方法可以总结如下:

方法 I: 使用恒等式 $f = \bar{\bar{f}}$ 的 SOP, 然后利用德摩根定律求出。

方法 II: 找出 \bar{f} 的 SOP 表示式的对偶形式, 然后对每一个变量取反。

图 2-8 展示了公式 (2-4) POS 表达式的或-与电路图。和与-或电路图类似, 或-与电路图是两级电路且不包括中间的非门。两个并联或门的中间输出将作为与门的输入信号进而产生 f 的值。



图 2-8 POS 表达式 $f = (x + \bar{y})(\bar{x} + y)$ 的与-或门级电路图

2. 用或非门实现 POS 表达式

用或非门替换或与电路中的非门、或门和与门可以产生一个等价的只有或非门的电路。这是基于德摩根第二定律 $\bar{x}\bar{y} = \overline{\bar{x} + \bar{y}}$ 得出的结论。考虑图 2-8 中的或-与电路。该电路可以通过以下步骤设计成只有或非门的电路:

1) 将一个二输入的或非门的输入连接起来, 作为与非门等价的或非门逻辑代替电路中的每一个非门, 如下图所示, 即 $y = \overline{x + x} = \bar{x}$ 。



2) 在中间信号 m 和 n 的两端加入两个非门, 如图 2-9 所示。这个操作并不会影响电路行为, 因为两个非门不会改变原始信号的值 (例如 $\overline{\bar{m}} = m$)。

3) 用或非门替换每一个或非门的电路组合, 如图 2-10 所示。一个或非门等价于一个或

门后面跟着一个非门的组合。

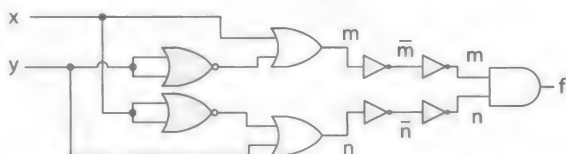


图 2-9 用加在线路两端的两个非门实现信号 m 和 n 的或 - 与门级电路图

唯一没有转变为或非门是有反相输入的与门(用内部非门标记)。根据德摩根第二定律 $\overline{m+n} = \overline{m}\overline{n}$ 可知这样的表示等同于一个或非门。

4) 用或非门代替有反相输入的与门的电路框图, 如图 2-11 所示。

以下的布尔代数可以表达成只用或非运算的 POS 表达式:

$$\begin{aligned} f &= (x + \bar{y})(\bar{x} + y) \\ &= \overline{\overline{(x + \bar{y})}(\bar{x} + y)} \\ &= \overline{(x + \bar{y}) + (\bar{x} + y)} \end{aligned} \quad (2-5)$$

逻辑项 $\overline{(x + \bar{y})}$ 和 $\overline{(\bar{x} + y)}$ 代表或非项, 通过或非运算生成 f 。概括地说, 一个函数的 SOP 和 POS 表达式是等价的, 且两者都可以生成相同的真值表。

所以, SOP 表达式用于与非门电路, POS 表达式用于或非门电路。

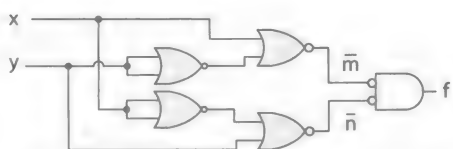


图 2-10 用内部非门实现的 POS 电路

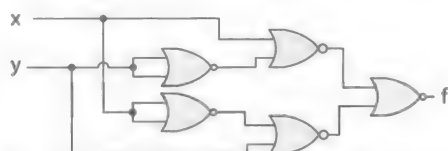


图 2-11 只用或非门实现 POS 表达式 $f = (x + \bar{y})(\bar{x} + y)$ 的电路图

2.3 规范表达式

当一个逻辑表达式中每一个逻辑项都包含了所有的输入变量或者其反相, 那么这个表达式就称为规范表达式(简称范式)。例如, 二变量函数 $f = \bar{x}\bar{y} + xy$ 是一个 SOP 范式。两个乘积项中都包含变量 x 和 y 或者它们的反相形式。同理, 二变量函数 $f = (x + \bar{y})(\bar{x} + y)$ 是一个 POS 范式。非规范表达式中可以包含一个或者多个不包含所有变量的逻辑项。例如, 三变量 SOP 表达式

$$g = \bar{x}\bar{y} + \bar{x}z + xyz$$

就不是一个范式, 因为逻辑 $\bar{x}\bar{y}$ 项少了 z 和 \bar{z} , 逻辑项 $\bar{x}z$ 少了 y 和 \bar{y} 。一个给定的非规范 SOP 表达式或者非规范 POS 表达式可以是或者不是极小的; 然而, 它可以先转换成等价的范式, 然后用以下的化简方法变成极小的范式。

2.3.1 极小项

乘积项的对应输入值称为极小项。例如, 考虑 SOP 范式 $f = \bar{x}\bar{y} + xy$ 有两个乘积项 $\bar{x}\bar{y}$ 和 xy 。 f 的两个乘积项对应的输入值为 $x = 0$ 且 $y = 0$, x 和 y 连接起来表示即 $(00)_2 = 0$ (原书有误——译者注), 或者当 $x = 1$ 且 $y = 1$, 即 $(11)_2 = 3$, 则 0 和 3 称为 f 的极小项, 可以用希腊符号 Σ (sigma) 表达如下:

$$f(x, y) = \Sigma(0, 3)$$

使用极小项能最直接地表达 SOP 范式的输出。如果极小项给出的形式是十进制数, 则需要先将它们转换为二进制数, 然后得出二进制数对应的乘积项, 如以下函数 g :

38

39

$$\begin{aligned}
 g(x,y,z) &= \Sigma(0,1,6,7) \\
 g(x,y,z) &= \Sigma((000)_2, (001)_2, (110)_2, (111)_2) \\
 g &= \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz
 \end{aligned}$$

对于一个输出变量来说, 其真值表、极小项列表和 SOP 范式是三种等价的表达方式。

2.3.2 极大项

同理, 和项的对应输入值称为极大项。极大项也可以写为整数形式, 用希腊符号 Π (π) 进行相乘得出 POS 表达式。每一个极大项都与 POS 范式中的一个和项相对应。例如, 表达式

$$f(x,y) = \Pi(0,1)$$

描述了 f 的极大项, 当 f 的两个输入 x 和 y 为 $(00)_2 = 0$ 或者 $(01)_2 = 1$ 时, f 为 0。

函数 f 的极大项就是其互补函数 \bar{f} 的极小项, 反之亦然。对于任意函数 h , 以下 1) ~ 3) 步说明了如何从其极大项列表获得其 POS 范式。步骤 i) 和 ii) 作为补充, 用于说明如何从 \bar{h} 的极小项列表中获得其 SOP 表达式。

40

- 1) $h(x,y,z) = \Pi(0,1,6,7)$
 $h(x,y,z) = \Pi(000,001,110,111)$
- 2) $h(x,y,z) = \overline{\bar{h}}$ 的 SOP 表达式
 $h(x,y,z) = \overline{\bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz}$
- 3) $h(x,y,z) = (\bar{x}\bar{y}\bar{z})(\bar{x}\bar{y}z)(xy\bar{z})(xyz)$
 h 的 POS 表达式 $(x,y,z) = (x+y+z)(x+y+\bar{z})(\bar{x}+\bar{y}+z)(\bar{x}+\bar{y}+\bar{z})$
- i) $\bar{h}(x,y,z) = \Sigma(0,1,6,7)$
 $\bar{h}(x,y,z) = \Sigma(000,001,110,111)$
- ii) $\bar{h}(x,y,z) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz$

步骤 2) 和 3) 也可以用对偶原理 (即方法 II) 代替, 如下所示:

将对偶原理应用到 \bar{h} 的 SOP 表达式中, 可得到其对偶表达式 $(\bar{x} + \bar{y} + \bar{z})(\bar{x} + \bar{y} + z)(x + y + \bar{z})(x + y + z)$; 然后将表达式中每一个变量取反, 则可得到 h 的 POS 表达式。

再次强调, 对于一个输出变量, 其真值表、极大项列表和 POS 范式是三种等价的表达。

2.4 逻辑化简

在第 1 章中已经讨论过, 以最少的逻辑门以及每个逻辑门以最少的输入来生成输出信号是非常重要的。最小的 SOP 或者 POS 表达式包含了最少数目的逻辑项, 每一项都是由最少数目的变量组成。图 2-12 展示了逻辑化简的优点。在这两个例子中, 规范表达式的电路实现需要更多的逻辑门, 即需要更多的晶体管和电线来实现。

在图 2-12 中, SOP 规范表达式的电路需要 4 个三输入的与门、3 个非门和 1 个四输入的或门或者总共需要 8 个与非门实现, 包括非门在内, 每一个逻辑门至多有 4 个输入。另一方面, 与其等价的最小 SOP 表达式只需要 2 个二输入的与门、1 个非门和 1 个二输入的或门, 或者一共需要 5 个与非门, 每一个逻辑门的输入都减少了。类似地, f 的规范 POS 表达式需要一共 8 个或非门, 而其最小表达式只需要 5 个或非门, 每一个逻辑门的输

入减少了。

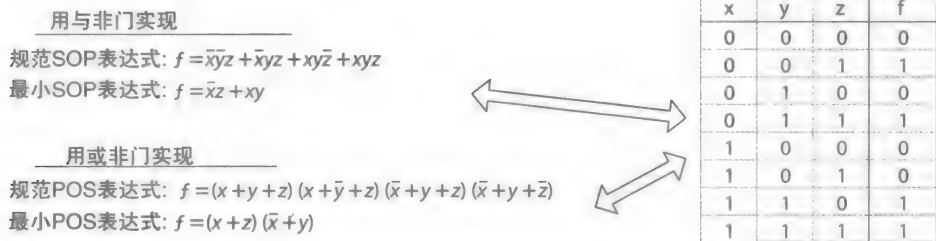


图 2-12 SOP 和 POS 最小表达式与规范表达式

41

正如之前讨论的, 规范 SOP 表达式或者规范 POS 表达式分别可以很容易地从最小项或者最大项推导而来。最小 / 最大项可以直接从真值得来。我们也可以根据给定的规范 SOP 表达式或者规范 POS 表达式写出其对应的真值表。然而, 不推荐也不必这样做。

确定给定非规范表达式的最小项或最大项不是一个简单的过程。不推荐通过判定每一个输入信号条件来确定其最小项或最大项。例如, 确定表达式 $f = y(\bar{x} + xz)$ 的真值表, 需要在所有 3 位输入 x 、 y 和 z 的各种取值组合下计算该表达式的值。例如, 由 $x = 0$ 、 $y = 1$ 和 $z = 0$ 可得 $f = 1$, 由 $x = 1$ 、 $y = 1$ 和 $z = 0$ 可得 $f = 0$, 等等, 这样可以确定整个真值表。然后可以从 f 的真值表得到其最小项和最大项。

还有一种选择是运用布尔代数把一个不规范的表达式转换成规范表达式。若表达式是一个 SOP 表达式, 则直接将规范表达式转换成其最小项; 若表达式是一个 POS 表达式, 则直接将规范表达式转换成其最大项。这个过程与逻辑化简步骤相反, 如下所示:

$$\begin{aligned}
 f &= y(\bar{x} + xz) && \text{不规范的表达式} \\
 f &= \bar{x}y + xyz && \text{分配 } y \\
 f &= \bar{x}y(z + \bar{z}) + xyz && \text{将缺少的 } z \text{ 和 } \bar{z} \text{ 插入到第一项中} \\
 f &= \bar{x}yz + \bar{x}y\bar{z} + xyz && \text{规范的 SOP 表达式}
 \end{aligned}$$

使用以下步骤将一个函数 f 的非规范 POS 表达式转换成与其对应的规范表达式:

- 1) 使用等式 “ \bar{f} 的 SOP 表达式 = $\overline{\bar{f}}$ 的 POS 表达式” 将 f 的非规范 POS 表达式转换成 \bar{f} 的非规范 SOP 表达式。
- 2) 通过 \bar{f} 的非规范 SOP 表达式求得其规范 SOP 表达式。
- 3) 使用等式 “ f 的 POS 表达式 = $\overline{\bar{f}}$ 的 SOP 表达式” 将 \bar{f} 的规范 SOP 表达式转换成 f 的规范 POS 表达式。

2.4.1 卡诺图

卡诺图 (K 图) 是一种识别和消除规范表达式中冗余项及获得一个或多个等价最小表达式的图形技术。两个用二进制表示仅有一位不相同的最小项或者最大项可以简化一个变量。例如, 三个变量 x 、 y 、 z , 两个最小项 $2 = (010)_2$ 和 $3 = (011)_2$ 中 z 的变量取值是不一样的。此项简化成 $\bar{x}y$ 的过程如下所示:

$$\bar{x}y\bar{z} + \bar{x}yz = \bar{x}y(\bar{z} + z) = \bar{x}y$$

$$\text{因为 } \bar{z} + z = 1 \text{ 和 } \bar{x}y \cdot 1 = \bar{x}y$$

类似地, 两个用二进制表示仅有一位不相同的最大项可以简化成减少一个变量的逻辑项。考虑以下由最大项 $2 = (010)_2$ 和 $3 = (011)_2$ 组成的规范 POS 表达式:

42

$$\begin{aligned}
 & (x + \bar{y} + z)(x + \bar{y} + \bar{z}) \\
 &= ((x + \bar{y}) + z)((x + \bar{y}) + \bar{z}) \\
 &= (x + \bar{y})(x + \bar{y}) + (x + \bar{y})\bar{z} + z(x + \bar{y}) + z\bar{z} \\
 &= (x + \bar{y}) + (x + \bar{y})(\bar{z} + z) + 0 \\
 &= (x + \bar{y}) + (x + \bar{y}) \\
 &= (x + \bar{y})
 \end{aligned}$$

组合较小逻辑项并分配
化简并提取公因子
化简
化简

在 K 图中, 任何两个用二进制表示仅有一位不相同的逻辑项互相连接, 最小 / 最大项以这种方式来组织, 使得定义这些逻辑项更容易。例如, 三个变量 x 、 y 、 z 有 8 种可能的最小 / 最大项组织成 2×4 或者 4×2 的 K 图, 分别如图 2-13a 和图 2-13b 所示。K 图中的每一个方格表示一个最小 / 最大项, 并且定义为以二进制表示的行标和列标的组合。例如, 0 行 00 列的方格定义了值为 0 的最小 / 最大项; 0 行 01 列的方格定义了值为 1 的最小 / 最大项; 等等。

在图中, 用二进制表示的相邻行标和列标只有一位不同。图 2-13a 中标记为 00 和 10 的两列也被认为是相邻的; 所以图 2-13b 中标记为 00 和 10 的两行也是相邻的。这样的定义有助于直观地看出只有一位不相同的最小 / 最大项。每一个逻辑项都与其上下左右四个方向的逻辑项相邻。例如, 在图 2-13a 中, 逻辑项 $0 = (000)_2$ 与其左方的逻辑项 $2 = (010)_2$ 和其右方的逻辑项 $1 = (001)_2$ 是相邻的, 与其下方的逻辑项 $4 = (100)_2$ 也是相邻的。逻辑项 $(000)_2$ 与 $(001)_2$ 、 $(010)_2$ 和 $(100)_2$ 有一位不相同。图 2-14 展示了一个四输入的 K 图。例如逻辑项 0, 分别与逻辑项 1 (右方)、2 (左方)、4 (下方) 和 8 (上方) 相邻。

函数 $g(x, y, z) = \Sigma(2, 6, 7)$ 的 K 图如下所示, 其中用 1 表示其每一个最小项。其余的方格对应于其最大项和留空项。

| yz | 00 | 01 | 11 | 10 |
|-----|----|----|----|----|
| x 0 | | | | 1 |
| 1 | | | 1 | 1 |

在 K 图中, 最小项 $2 = (010)_2$ 与另一个最小项 $6 = (110)_2$ 相邻, 而最小项 6 与最小项 $7 = (111)_2$ 相邻。然而最小项 2 与最小项 7 并不相邻, 二进制表示 $(010)_2$ 与 $(111)_2$ 有两位不同。函数 $g(x, y, z) = \Pi(0, 1, 3, 4, 5)$ 的 POS 表达式的 K 图如下所示:

| yz | 00 | 01 | 11 | 10 |
|-----|----|----|----|----|
| x 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | | |

| yz: | 00 | 01 | 11 | 10 |
|------|----|----|----|----|
| x: 0 | 0 | 1 | 3 | 2 |
| 1 | 4 | 5 | 7 | 6 |

a) 2×4 结构

| z: | 0 | 1 |
|--------|---|---|
| xy: 00 | 0 | 1 |
| 01 | 2 | 3 |
| 11 | 6 | 7 |
| 10 | 4 | 5 |

b) 4×2 结构

图 2-13 一个三变量的 K 图结构, 每一个方格代表一个最小 / 最大项

| yz: | 00 | 01 | 11 | 10 |
|--------|----|----|----|----|
| wx: 00 | 0 | 1 | 3 | 2 |
| 01 | 4 | 5 | 7 | 6 |
| 11 | 12 | 13 | 15 | 14 |
| 10 | 8 | 9 | 11 | 10 |

图 2-14 一个四变量的 K 图, 方格中的数为与其对应的最小 / 最大项的值

2.4.2 K 图化简

K 图可以产生化简后的表达式。一个函数可以有多个化简后的表达式，它们之间是等价的。一对相邻的逻辑项可以化简成减少一个变量的逻辑项，多个相邻的逻辑项可以化简成减少更多变量的逻辑项。如图 2-15 所示，K 图中有 4 个最小项，分别为 2、3、6 和 7。

| | | | | |
|------|----|----|----|----|
| yz: | 00 | 01 | 11 | 10 |
| x: 0 | | | 1 | 1 |
| 1 | | | 1 | 1 |

图 2-15 K 图中有 4 个最小项，分别为 2、3、6、7

最小项 $(010)_2$ 与最小项 $(011)_2$ 和 $(110)_2$ 相邻，最小项 $(011)_2$ 与最小项 $(010)_2$ 和 $(111)_2$ 相邻，最小项 $(110)_2$ 与 $(010)_2$ 和 $(111)_2$ 相邻。运用布尔代数，这些最小项可以化简为 y ，如下所示：

$$\begin{aligned}
 \Sigma(2,3,6,7) &= \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz \\
 &= \bar{x}y(\bar{z} + z) + xy(\bar{z} + z) && \text{提取较小逻辑项并化简} \\
 &= \bar{x}y + xy && \text{提取 } y \text{ 并化简} \\
 &= y(\bar{x} + x) && \text{化简} \\
 &= y
 \end{aligned} \tag{2-6}$$

例如公式 (2-6) 和图 2-15 的化简，运用 K 图化简步骤如下：

1) 比较两个列标 11 和 10，其中与其有联系的是最小项组 $(010)_2$ 、 $(011)_2$ 、 $(110)_2$ 和 $(111)_2$ 。代表变量 z 的位标记改变了，而代表变量 y 的位标记没有变化；这样变量 z 可以消掉。(在代数运算中，较小项 $\bar{x}y$ 和 xy 被提出来，从而消掉 z 。)

2) 比较两个行标，信号 x 变化了，而与其有位联系的 y (剩下的变量) 没有变化；这样， x 可以被消掉。(在代数运算中， y 可以被提取出来从而消掉 x 。)

3) 写下没有被消掉的变量 (在这个例子中只有 y) 作为前面第 1) 和第 2) 步的化简结果，当位标记为 0 时信号为 \bar{y} ，当位标记为 1 时信号为 y 。这个例子中 y 为最小项组的最终化简结果，在公式 (2-6) 中也有相应的代数运算。

运用 K 图，无需进行布尔运算就可以求出与规范表达式等价的最小化简的表达式。考虑下列有最小项 1、2、3、6 和 7 的 K 图：

| | | | | |
|------|----|----|----|----|
| yz: | 00 | 01 | 11 | 10 |
| x: 0 | | 1 | 1 | 1 |
| 1 | | | 1 | 1 |

相邻的最小项组 2、3、6 和 7 可以被化简为 y 。对于最小项 $1 = (001)_2$ 来说，唯一与其相邻的最小项是 $3 = (011)_2$ 。比较两个列标 01 和 11，变量 y 有改变，所以可以被消掉，得到结果逻辑项为 $\bar{x}z$ ，对应为 $x = 0$ 和 $z = 1$ 。这样，最小项 1、2、3、6 和 7 最终的最小项化简表达式为

$$y + \bar{x}z \tag{2-7}$$

注意到最小项 3 运用了两次，一次是在最小项组 2、3、6 和 7 中，一次在最小项组 1 和 3 中。这个重复运用的依据是布尔代数中的规则 $x = x + x$ (或者对于最大项， $x \cdot x = x$)。在逻辑表达式中重复使用逻辑项并不会改变其真值表，但是却可以帮助相邻逻辑项组成更大的逻辑项组，从而在最终的化简表达式中消掉更多的变量。下列布尔运算展示了这一点：

给定一个规范的 SOP 表达式 $\bar{x}\bar{y}z + \bar{x}yz + \bar{x}yz + xy\bar{z} + xyz$ ，重复使用对应于最小项 3 的逻辑项 $\bar{x}yz$ ：

$$(\bar{x}\bar{y}z + \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz) + \bar{x}yz$$

使用相邻的原则重组逻辑项:

$$(\bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz) + (\bar{x}\bar{y}z + \bar{x}yz)$$

化简每一组逻辑项, 最终得到化简的最小 SOP 表达式 $y + \bar{x}z$ 。

如果没有 K 图, 决定一个逻辑项需要重复几次是非常困难的。最小表达式是通过 K 图中每一组相邻的逻辑项的最大数目得来的。K 图的化简规则总结如下。

1. K 图化简规则

1) 我们把只有一位不同的最小 / 最大项称为是相邻的, 并且它们被认为形成一个蕴含。我们假定 K 图是两侧都可以环绕的, 即 00 列和 10 列也相邻。

2) 一组蕴含可以组合起来形成一个称为素蕴含的大组。每一个组包含的逻辑项数目都是 2 的幂, 即一个组可能含有一个逻辑项、两个逻辑项、4 个逻辑项或者 8 个逻辑项。

3) 每一个素蕴含必须至少包含一个不属于其他任何素蕴含的单独的逻辑项 (例如无冗余组)。符合这个规则的素蕴含被称为基本素蕴含 (EPI)。最终的最小表达式必须包含所有基本素蕴含的逻辑项。

4) 所有逻辑项都必须分组。

例 2-3 化简表达式 $f(x, y, z) = \Sigma(1, 3, 6, 7)$ 。

解: f 的 K 图如下:

| yz | 00 | 01 | 11 | 10 |
|-----|----|----|----|----|
| x 0 | | 1 | 1 | |
| 1 | | | 1 | 1 |

最小项 1、3、6 和 7 组成了三个而不止一个素蕴含, 如 K 图所示。这是因为最小项 $1 = (001)_2$ 和 $6 = (110)_2$ 都只与其他一个逻辑项相邻。由最小项 $3 = (011)_2$ 和 $7 = (111)_2$ 组成的素蕴含不是一个 EPI。因为有一个素蕴含包含了最小项 1 和 $3 = (011)_2$, 变量 y 的列标改变了, 得到逻辑项 $\bar{x}z$ 。由最小项 6 和 7 组成的素蕴含, 变量 z 的列标改变了, 得到逻辑项 xy 。即最终的最小 SOP 表达式为:

$$f = \bar{x}z + xy$$

例 2-4 化简表达式 $f(x, y, z) = \Sigma(0, 1, 2, 3, 4, 6)$ 。

解:

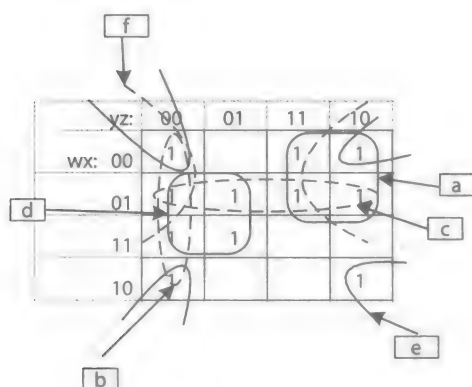
| yz | 00 | 01 | 11 | 10 |
|-----|----|----|----|----|
| x 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | | | 1 |

4 个角的最小项 0、2、4 和 6 与第一行的最小项 0、1、2 和 3 组成了两个 EPI。由于由最小项 0、1、2 和 3 组成的素蕴含中变量 y 和 z 改变了, 所以消掉这两个变量得到逻辑项 \bar{x} 。类似地, 由于由最小项 0、2、4 和 6 组成的素蕴含中变量 x 和 y 改变了, 所以得到逻辑项 \bar{z} 。最终的最小 SOP 表达式为:

$$f(x, y, z) = \bar{x} + \bar{z}$$

例 2-5 化简表达式 $f(w, x, y, z) = \Sigma(0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13)$ 。

解：



图中有 5 个素蕴含和它们对应的最小项列表。只有标记为 a、d 和 e 的素蕴含为 EPI。因为素蕴含 a、d 和 e 都只包含了素蕴含 b、c 和 f 中的部分最小项。

$$a: \Sigma(2,3,6,7)$$

$$b: \Sigma(0,4,8,12) \quad \text{冗余}$$

$$c: \Sigma(4,5,6,7) \quad \text{冗余}$$

$$d: \Sigma(4,5,12,13)$$

$$e: \Sigma(0,2,8,10)$$

$$f: \Sigma(0,2,4,6) \quad \text{冗余}$$

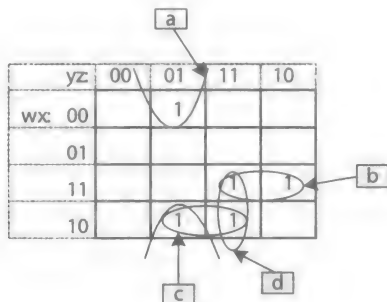
对于素蕴含 a，变量 x 和 z 的标记是有变化的，所以得到最小项 $\bar{w}y$ ；对于素蕴含 d，变量 w 和 z 的标记是有变化的，所以得到 $x\bar{y}$ ；对于蕴含 e，变量 w 和 y 的标记有变化，所以得到 $\bar{x}\bar{z}$ 。最后得到的最小 SOP 表达式为：

$$f(w, x, y, z) = \bar{w}y + x\bar{y} + \bar{x}\bar{z}$$

47

例 2-6 化简表达式 $f(w, x, y, z) = \Sigma(1, 9, 11, 14, 15)$ 。

解：



素蕴含含有：

$$a: \Sigma(1,9)$$

$$b: \Sigma(14,15)$$

$$c: \Sigma(9,11)$$

$$d: \Sigma(11,15)$$

素蕴含 a、b 以及 c 或 d 中的一个为 EPI。EPI a、b 和 c 化简后可得到下列最小 SOP 表达式：

$$f(w, x, y, z) = \bar{x}\bar{y}z + wxy + w\bar{x}z$$

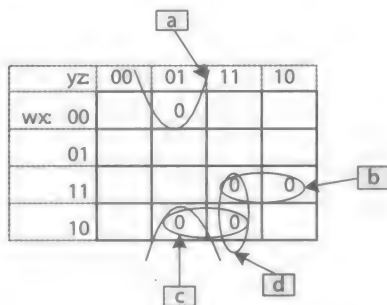
类似地，用 EPI a、b 和 d 可以得到等价的最小 SOP 表达式：

$$f(w, x, y, z) = \bar{x}\bar{y}z + wxy + wyz$$

两个最小表达式是等价的，因为它们的真值表是一样的，拥有相同的最小 / 最大项。 ■

例 2-7 化简表达式 $g(w, x, y, z) = \Pi(1, 9, 11, 14, 15)$

解：



POS 表达式的素蕴含和 SOP 表达式的素蕴含作用是一样的，然而在这个例子中，表达式 g 的 K 图中的 0，等价于表达式 \bar{g} 的 K 图中的 1，这些 0 可以组成逻辑项组，使用之前讨论的方法 (f 的 POS 表达式 = \bar{f} 的 SOP 表达式) 决定 g 的 POS 表达式。在此例中，与例 2-6 相似，表达式 g 的 SOP 素蕴含含有 $\Sigma(1, 9) = \bar{x}\bar{y}z$, $\Sigma(14, 15) = wxy$, $\Sigma(9, 11) = w\bar{x}z$ 和 $\Sigma(11, 15) = xyz$ 。这样，可以得到表达式 g 的 POS 素蕴含为：

$$a: \Pi(1, 9) = \bar{x}\bar{y}z = (x + y + \bar{z})$$

$$b: \Pi(14, 15) = wxy = (\bar{w} + \bar{x} + \bar{y})$$

$$c: \Pi(9, 11) = w\bar{x}z = (\bar{w} + x + \bar{z})$$

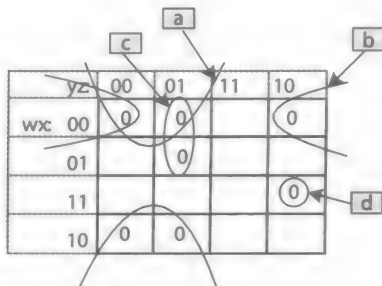
$$d: \Pi(11, 15) = xyz = (\bar{w} + \bar{y} + \bar{z})$$

类似地，表达式 g 有两个最小 POS 表达式：1) 使用 EPI a、b 和 c；2) 使用 EPI a、b 和 d。使用 2) 可得到下列 POS 表达式：

$$g = (x + y + \bar{z})(\bar{w} + \bar{x} + \bar{y})(\bar{w} + \bar{y} + \bar{z})$$

例 2-8 化简表达式 $f(w, x, y, z) = \Pi(0, 1, 2, 5, 8, 9, 14)$ 。

解：



素蕴含含有：

- a: $\Pi(0,1,8,9)$
- b: $\Pi(0,2)$
- c: $\Pi(1,5)$
- d: $\Pi(14)$

这些素蕴含都是基本素蕴含，且它们对应的最小 POS 表达式如下：

- a: 变量 w 和 z 的标记有改变，得到 $(x + y)$ 。
- b: 变量 y 的标记有改变，得到 $(w + x + z)$ 。
- c: 变量 x 的标记有改变，得到 $(w + y + \bar{z})$ 。
- d: 此蕴含只包含一个逻辑项，得到 $(\bar{w} + \bar{x} + \bar{y} + z)$ 。

所以最终的最小表达式为：

$$f(w,x,y,z) = (x + y)(w + x + z)(w + y + \bar{z})(\bar{w} + \bar{x} + \bar{y} + z)$$

2. 无关项

在极少一些情况下，输出可能只由当前输入条件的一个子集决定，而其他的输入条件是不确定的。例如，考虑图 2-16 中的 7 段显示单元 (7SDU) 及其转换模块。假设转换器用于显示 0 ~ 9 的二 - 十进制编码数字 (BCD)。给定 4 位输入，范围为 $0 = (0000)_2$ 到 $9 = (1001)_2$ ，转换器生成了 7 个信号，从 fa 到 fg ：一个信号可以使得 a 到 g 的 7 段中的一段显示管开启，使其对应 0 ~ 9 的数字。

例如，要显示数字 0，除了 g 段所有的显示管都必须开启；这样，除了信号 fg ，所有的信号值都必须为 1 (假设使用高电平输出)。如果要显示数字 9，除了信号 ff ，所有的信号必须开启，除了 ff ，所有信号值必须为 1。当输入信号为 4 位 0 ~ 9 的数字时，BCD 到 7SDU 转换器用于正确输出 fa 到 fg 信号。当输入信号为 10 ~ 15 时，输出是没有定义的，可以被认为是无关的，在真值表中以 d 表示。

一个无关的最小 / 最大项可以用“通配符”表示，根据 K 图的需要可以表示为 0 或者 1。这样，无关项就可以用于帮助消掉变量和化简最终的表达式。考虑表达式 $f(w,x,y,z) = \Sigma(1,9,14) + \Sigma_d(3,7,11)$ ，其中， Σ_d 用于表示表达式 f 的最小项中的无关项。类似地，符号 Π_d 用于表示函数中最大项中的无关项。如下所示，其中一个基本蕴含包含了两个无关项，只有无关项被用到来化简表达式。无关项变成的最小项 3 和 11 和最小项 1 和 9 一起生成了基本蕴含 $\bar{x}z$ 。最小项 7，不是一个无关项，不需要用到。所以最后最小化简的 SOP 表达式为 $f(w,x,y,z) = \bar{x}z + wxy\bar{z}$ 。

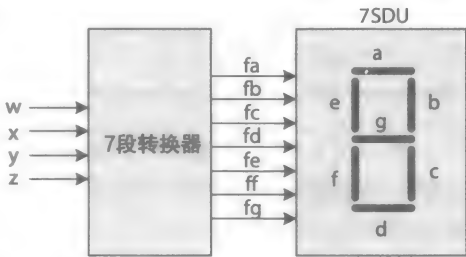


图 2-16 7 段显示器及其转换器

| yz | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| wc | | 1 | d | |
| | | | d | |
| | | | | ① |
| | | | | |
| | | 1 | d | |

2.5 逻辑化简算法

K 图化简是一种图形的方式，只能适用于很小数目的变量，例如 4 个变量的表达式中。当

变量数大于 4 时，在 20 世纪 50 年代中期发明的叫作奎因－麦克拉斯基算法的数学方式更合适于表达式化简。这个算法使用比 K 图更少的步骤来寻找最小逻辑表达式。最小项被划分成不同的集合，每一个集合都只包含一个在二进制表达中有特殊个 1 的个数的最小项。考虑例 2-5 中的表达式 $f(w, x, y, z) = \Sigma (0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13)$ 。在二进制中，最小项为 0000, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1010, 1100 和 1101。这些最小项可以分组成下列 4 个集合：

| | | |
|-------|------|-------------|
| 集合 1: | 0000 | 不包含 1 的最小项 |
| 集合 2: | 0010 | 包含一个 1 的最小项 |
| | 0100 | |
| | 1000 | |
| 集合 3: | 0011 | 包含两个 1 的最小项 |
| | 0101 | |
| | 0110 | |
| | 1010 | |
| | 1100 | |
| 集合 4: | 0111 | 包含三个 1 的最小项 |
| | 1101 | |

每一次比较一对最小项，集合 1 中的一个最小项与集合 2 中的一个最小项比较。在每一对最小项比较中，一位数字的改变意味着一个蕴含。改变位可以用一条横杠 (—) 表示，从而省略对应的变量。

例如，集合 1 中的最小项 0000 与集合 2 中的 0010 比较，可以生成蕴含 00—0，变量 y 被省略。最小项 0000 可以再与最小项 0100 比较，生成蕴含 0—00，变量 x 被省略，等等。这个过程（每一次比较一对最小项）可以用于所有最小项。集合 2 和集合 3，集合 3 和集合 4 可以生成用于结果的一个蕴含的集合。初始集合 1 到集合 4 标记为表 2-4 列 I 中的 I.1 到 I.4。

使用集合 I.1 到 I.4 生成的蕴含列表标记为集合 II.1 到 II.3，写在表 2-4 的列 II 中。接下来，对于集合 I.1 到 I.4 中的最小项，如果在列 II 的最小项可以生成一个蕴含，就用一个“ x ”标记；否则，就用星号 (*) 标记，识别出一个素蕴含（在这个例子中没有）。当列 II 中的蕴含生成之后，重复之前的步骤；每一次比较一对集合 II.1 和集合 II.2 中的一对蕴含。在这个例子中，在每一对蕴含之间的横杠必须排列起来。

51

例如，集合 II.1 中的蕴含 00—0 与集合 II.2 中的蕴含 10—0 比较。生成的蕴含写在表中的列 III 下的集合 III.1 和 III.2。同样，如果列 II 中的蕴含对生成的不是素蕴含，用“ x ”标记；否则，用星号 (*) 标记，在这个例子中没有素蕴含。这个过程在列 III 中重复一次，但是这次，没有更多的步骤可以走了，因为集合 III.1 和集合 III.2 中的蕴含——比较之后，不能再生成新的蕴含；这样，所有从集合 III.2 和集合 III.3 中产生的蕴含都标记为星号 (*)，与其对应的逻辑项在图 2-17 中列出。

表 2-4 最小项 0, 2, 3, 4, 5, 6, 7, 8, 10, 12 和 13 生成的素蕴含（用 * 标记）列表

| | 列 I | | | | | | 列 II | | | | | | 列 III | | | | |
|---------|-----|-----|-----|-----|-----|----------|------|-----|-----|-----|-----|-----------|-------|-----|-----|-----|---|
| | w | x | y | z | | | w | x | y | z | | | w | x | y | z | |
| 集合 I.1: | 0 | 0 | 0 | 0 | x | 集合 II.1: | 0 | 0 | — | 0 | x | 集合 III.1: | — | 0 | — | 0 | * |
| | | | | | | | 0 | — | 0 | 0 | x | | 0 | — | — | 0 | * |

(续)

| | 列 I | | | | | | 列 II | | | | | | 列 III | | | | |
|---------|-----|---|---|---|----------|----------|------|---|---|---|----------|-----------------|-------|---|---|---|---|
| 集合 I.2: | 0 | 0 | 1 | 0 | <i>x</i> | | — | 0 | 0 | 0 | <i>x</i> | | — | — | 0 | 0 | * |
| | 0 | 1 | 0 | 0 | <i>x</i> | | | | | | | | | | | | |
| | 1 | 0 | 0 | 0 | <i>x</i> | 集合 II.2 | 0 | 0 | 1 | — | <i>x</i> | 集合 III.2: | 0 | — | 1 | — | * |
| | | | | | | | 0 | — | 1 | 0 | <i>x</i> | | 0 | 1 | — | — | * |
| 集合 I.3: | 0 | 0 | 1 | 1 | <i>x</i> | | — | 0 | 1 | 0 | <i>x</i> | | — | 1 | 0 | — | * |
| | 0 | 1 | 0 | 1 | <i>x</i> | | 0 | 1 | 0 | — | <i>x</i> | | | | | | |
| | 0 | 1 | 1 | 0 | <i>x</i> | | 0 | 1 | — | 0 | <i>x</i> | | | | | | |
| | 1 | 0 | 1 | 0 | <i>x</i> | | — | 1 | 0 | 0 | <i>x</i> | <i>x</i> : 非素蕴含 | | | | | |
| | 1 | 1 | 0 | 0 | <i>x</i> | | 1 | 0 | — | 0 | <i>x</i> | *: 素蕴含 | | | | | |
| | | | | | | | 1 | — | 0 | 0 | <i>x</i> | | | | | | |
| | 0 | 1 | 1 | 1 | <i>x</i> | | | | | | | | | | | | |
| 集合 I.4: | 1 | 1 | 0 | 1 | <i>x</i> | 集合 II.3: | 0 | — | 1 | 1 | <i>x</i> | | | | | | |
| | | | | | | | 0 | 1 | — | 1 | <i>x</i> | | | | | | |
| | | | | | | | — | 1 | 0 | 1 | <i>x</i> | | | | | | |
| | | | | | | | 0 | 1 | 1 | — | <i>x</i> | | | | | | |
| | | | | | | | 1 | 1 | 0 | — | <i>x</i> | | | | | | |

程序的下一步是要在素蕴含列表 a 到 f 用一个最小集合算法来选择基本蕴含。图 2-17 是素蕴含和它们对应最小项的组织图。如果一个素蕴含包含一个最小项，则在方格中用 “x” 标记。例如，有两个横杠的素蕴含 0 — 0 包含了最小项 0 = (0000)₂, 2 = (0010)₂, 4 = (0100)₂ 和 6 = (0110)₂, 这些最小项方格中在行 1 中标记为 “x”，如表格所示。

52

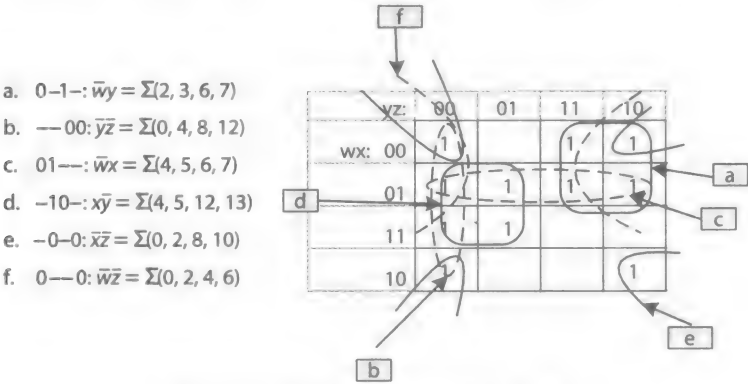


图 2-17 表 2-4 中包含的素蕴含列表

最小集合算法也是一个迭代的过程，最开始在任意列中选择只有一个有 “x” 标记的素蕴含，这样可以生成一个 EPI。在这个例子中，最小项 3、10 和 13 相关的列中只有一个 “x” 标记；这些类似的例子在表格中用粗体和下划线表示出来。

假设在表 2-5 中从左往右处理最小项。在迭代 1 中，最小项 3 相关的列只包含一个 “x”，这样对应的素蕴含 0 - 1 - 可以作为一个 EPI，因为其只包含最小项 3。它也可以包含最小项 2、3、6 和 7，这样相关的列和行被标记为删除 (D)，如表格中所示。这个过程可以有效地消掉下一个迭代中表格的规模。在迭代 2 中，素蕴含 - 0 - 0 在最小项 10 相关的列中

只有一个“x”标记，我们选择其为下一个 EPI。这样，列 0、8 和 10，行 2 标记为 *D*。最后，在迭代 3 中，与最小项 13 相关的素蕴含 - 10 -，被选作为下一个 EPI，这样则需要删掉表中所有的剩余列，以及 EPI = - 10 - 对应的行。

表 2-5 对表 2-4 中包含的素蕴含使用最小集合算法图解

| | 素蕴含 | | | | 最小项 | | | | | | | | | | | 迭 代 | | |
|-------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| | <i>w</i> | <i>x</i> | <i>y</i> | <i>z</i> | 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 12 | 13 | 1: | 2: | 3: |
| | 0 | - | - | 0 | <i>x</i> | <i>x</i> | | <i>x</i> | | <i>x</i> | | | | | | | | |
| EPI | - | 0 | - | 0 | <i>x</i> | <i>x</i> | | | | | | <i>x</i> | <i>x</i> | | | | <i>D</i> | |
| | - | - | 0 | 0 | <i>x</i> | | | <i>x</i> | | | | <i>x</i> | | <i>x</i> | | | | |
| | 0 | 1 | - | - | | | | <i>x</i> | <i>x</i> | <i>x</i> | <i>x</i> | | | | | | | |
| EPI | 0 | - | 1 | - | | <i>x</i> | <i>x</i> | | | <i>x</i> | <i>x</i> | | | | | <i>D</i> | | |
| EPI | - | 1 | 0 | - | | | | <i>x</i> | <i>x</i> | | | | | <i>x</i> | <i>x</i> | | | <i>D</i> |
| 迭代 1: | | | | | | <i>D</i> | <i>D</i> | | | <i>D</i> | <i>D</i> | | | | | | | |
| 迭代 2: | | | | | <i>D</i> | | | | | | | <i>D</i> | <i>D</i> | | | | | |
| 迭代 3: | | | | | | | | <i>D</i> | <i>D</i> | | | | | <i>D</i> | <i>D</i> | | | |

53

当所有与最小项有关的列都删除之后，算法结束。在这个例子中，经过三轮迭代后算法结束，并产生三个 EPI，- 0 - 0、0 - 1 - 和 - 10 -，“迭代”部分对应的行标记为 *D*。EPI 生成了最小 SOP 表达式 $\bar{x}\bar{z} + \bar{w}y + x\bar{y}$ ，在例 2-6 中，我们使用 K 图的方法也得到了这个表达式。

如果在最小集合算法过程中，没有发现有一列只有一个“x”标记，以下的规则可以用来选择下一个候选的素蕴含：

- 1) 找到至少有一个“x”标记的列，然后选择其对应的素蕴含作为候选项。
- 2) 在第 1) 步素蕴含列表中选择那些在剩下的最多最小项的素蕴含（不包括有 *D* 标记的列）。
- 3) 如果在第 2) 步中得到多个素蕴含，选择那个包含横杠最多的素蕴含；其对应的逻辑项应该包含较少变量。
- 4) 如果在第 3) 步中得到多个素蕴含，则可得到两个或以上的等价最小表达式。

除了在表 2-4 和表 2-5 中需要用最大项来替换最小项求最小 POS 表达式的算法过程也是相似的。

如果一个逻辑电路有多个输出，每个输出的最小表达式通常不会独立地确定。相反，此时化简的目标是选择在不同表达式之间相同的素蕴含，从而可以减少实现多个表达式电路中所需的逻辑门数目。一些逻辑门的输出信号可以被共享和连接为其他逻辑门的输入。Espresso 优化软件 [1] 可以解决需要同时化简多个逻辑表达式的情况。逻辑设计的 CAD 工具通常包含这个软件和其他化简软件。

化简软件

例 2-9 展示了用函数 $f(w, x, y, z) = \Sigma (0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13)$ 最小项来进行 Espresso 操作的输入和输出文件。第一列中的点 (·) 代表一个参数。例如，“.i 4”表示输入的个数，在这个例子中为 4，“.o 1”表示输出的位数，这个例子中为 1。标记 “.ilb w x y z”列出输入的变量，这个例子中为 4 个变量；“.ob f”列出输出变量，在这个例子中为 1 个变量；“.e”表示输入文件的结尾。符号 “#” 表示一行注释。在输出文件中，“.p”表示

EPI 的数目。

54

例 2-9 用 Espresso 化简函数 $f(w, x, y, z) = \Sigma (0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13)$ 。

解：

a) 输入文件

```
#Inputs: 4, Outputs: 1
.i 4
.o 1
#Input labels
.ilb w x y z
#output bit label
.ob f
#list of min-terms separated by space and a single output
bit separated by a tab
0 0 0 0 1
0 0 1 0 1
0 0 1 1 1
0 1 0 0 1
0 1 0 1 1
0 1 1 0 1
0 1 1 1 1
1 0 0 0 1
1 0 1 0 1
1 1 0 0 1
1 1 0 1 1
#end of list
.e
```

b) 输出文件：所有注释行首先打印

```
#Inputs: 4, Outputs: 1
#Input signal labels
#output bit label
#list of min-terms and output
#end of list
.i 4
.o 1
.ilb w x y z
.ob f
.p 3
-10- 1
-0-0 1
0-1- 1
.e
```

55

输出文件列出了 3 个 EPI：-10-、-0-0 和 0-1-。这 3 个 EPI 与之前手工步骤求出的相同，在表 2-4 和表 2-5 中列出。

下列的 Espresso 输出列出了两个输出信号 f 和 g 的 EPI。“11”、“11”和“10”打印在 EPI 之后，这三个 EPI 都是属于输出信号 f 的，而前两个（-10- 和 -0-0）是共享的并属于输出变量 g 。

```
.o 2
.ob f g
.p 3
-10- 11
-0-0 11
0-1- 10
.e
```

表 2-6 展示了表达式 $f(w, x, y, z) = \Sigma (1, 9, 14) + \Sigma_d (3, 7, 11)$ ，运用化简算法后的结果，这个表达式也在 2.4.2 小节中化简过。在列 I 中，每一个无关项都标记为“d”。之前讨论过的化简算法也是一样的，只是在一对最小项中只能有一个无关项；两个无关项不能进行比较。例如，集合 I.2 中最小项 $(0011)_2$ 和集合 I.3 中的最小项 $(0111)_2$ ，两个都为无关项，不能进行比较来产生素蕴含 $0-11$ 。

算法产生三个素蕴含（以星号为标记），每一个在一行中。表 2-7 是表 2-6 运用最小集合算法之后的素蕴含组织表。集合 I.3 素蕴含 $(1110)_2$ 是一个 EPI。在剩下的两个素蕴含 $-0-1$ 和 -001 中都包含最小项 1 和 9，由于 $-0-1$ 有比 -001 更多的横杠，所以选择 $-0-1$ 。这些 EPI 也可以在例 2-10 中用 Espresso 算法实现，输入文件中的横杠（-）代表无关的候选项。

表 2-6 $f(w, x, y, z) = \Sigma (1, 9, 14) + \Sigma_d (3, 7, 11)$ 的素蕴含

| | I | | | | | | II | | | | | | III | | | | |
|---------|---|---|---|---|---|----------|---------|---|---|---|---|-----------|-----|---|---|---|---|
| | w | x | y | z | | | w | x | y | z | | | w | x | y | z | |
| 集合 I.1: | 0 | 0 | 0 | 1 | x | 集合 II.1: | 0 | 0 | — | 1 | x | 集合 III.1: | — | 0 | — | 1 | * |
| | | | | | | | — | 0 | 0 | 1 | * | | | | | | |
| 集合 I.2: | 0 | 0 | 1 | 1 | d | | | | | | | | | | | | |
| | 1 | 0 | 0 | 1 | x | 集合 II.2: | 1 | 0 | — | 1 | x | 集合 III.2: | | | | | |
| | | | | | | | | | | | | | | | | | |
| | 0 | 1 | 1 | 1 | d | | x: 非素蕴含 | | | | | | | | | | |
| 集合 I.3: | 1 | 0 | 1 | 1 | d | | *: 素蕴含 | | | | | | | | | | |
| | 1 | 1 | 1 | 0 | * | | | | | | | | | | | | |

56

表 2-7 运用表 2-6 中素蕴含进行的最小集合算法过程

| | w | x | y | z | 1 | 9 | 14 | 1: | 2: |
|-----|---|---|---|---|---|---|----|----|----|
| EPI | 1 | 1 | 1 | 0 | | | x | D | |
| | — | 0 | 0 | 1 | x | x | | | |
| EPI | — | 0 | — | 1 | x | x | | | D |
| 1: | | | | | | | D | | |
| 2: | | | | | D | D | | | |

例 2-10 用 Espresso 算法化简 $f(w, x, y, z) = \Sigma (1, 9, 14) + \Sigma_d (3, 7, 11)$ 。

(a) 输入文件

```
.i 4
.o 1
.ilb w x y z
.ob f
```

```
0 0 0 1 1
0 0 1 1 -
0 1 1 1 -
1 0 0 1 1
1 0 1 1 -
1 1 1 0 1
.e
```

(b) 输出文件

```
.i 4
.o 1
.ilb w x y z
.ob f
.p 2
1110      1
-0-1      1
.e
```

2.6 电路时序图

在讨论电路时序图之前，我们首先从非门电路时序图开始。每一个逻辑门都有一定的延时。这个延时是由于逻辑门输出从 0 到 1 或者反过来一个或多个输入变化需要时间。图 2-18 展示了非门 0.1ns 延迟的时序。如图 2-18a 所示，当输入 x 从 0 变为 1 时，输出 z 在 0.1ns 中从 1 变为 0。类似地，当 x 从 1 变为 0 时，在 0.1ns 中从 0 变为 1。在图 2-18a 中，信号的转换显示成立即发生。然而，在实际中，信号转换并非立即发生。

57

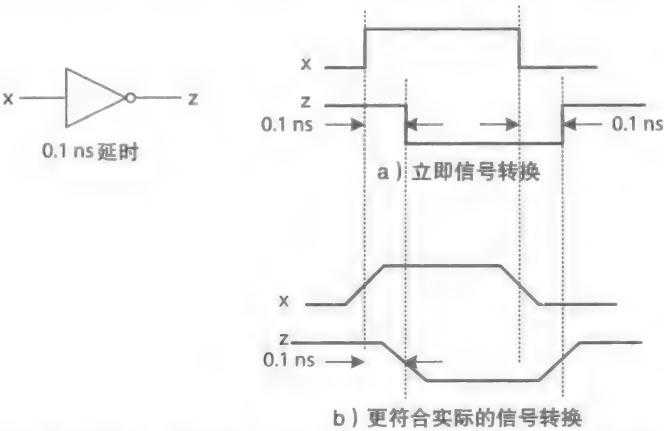


图 2-18 有 0.1ns 延时的非门时序图：a) 一个简单的时序图；b) 一个更符合实际的时序图

信号上升时间是输出电压从逻辑 0 的对应值上升到逻辑 1 的对应值所需的时间。类似的，信号下降时间是输出电压从逻辑 1 的对应值下降到逻辑 0 的对应值所需的时间。逻辑门的上升时间和下降时间可能不一样。图 2-18b 展示了更实际的非门时序电路图，包括其上升时间和下降时间。如图所示，上升和下降的中点通常用于现实信号的立即转换。

对于 + 5.0 伏特电压的电源，所有在 0 ~ 0.8V 之间的输入值都被视为逻辑 0，所有在 2.0 ~ 5.0V 之间的输入值都被视为逻辑 1。一个输入电压大于 0.8V 且小于 2.0V 之间的值都

被视为未定义的。对于逻辑 1 输出，电压范围是 2.4 ~ 5.0V 之间，对于逻辑 0，对应输出电压范围是 0 ~ 0.4V 之间。更低的电压源（例如，1.8V 或者 1.2V）通常用于电池供电的系统中。

最小布尔表达式定义了不需要考虑逻辑门和线路延迟的电路输入和输出之间的逻辑关系。一个电路时序图是每个逻辑门的输出在电路中由于逻辑门和电路延迟而发生实际变化的说明图。当电路输入发生变化时，时序图提供了电路行为的一个更实际的视图。例如，考虑表达式 $f(a, b, c, d) = \Sigma(1, 3, 5, 7, 10, 11, 14, 15)$ ，其最小 SOP 表达式为 $f = \bar{a}d + ac$ ；由此可看出 f 不由 b 决定。图 2-19 展示了与其等价的与非门电路，中间信号为 \bar{a} 、 x 和 y ；逻辑门被标记为 G1 到 G4。

图 2-20 展示了当输入从 $acd = 111$ 到 $acd = 011$ 时电路的时序图；即 a 从 1 转变为 0。假设所有逻辑门的延迟为 0.1ns，线路延迟忽略不计。（线路延迟超出本书范围。）考虑到当输入为 $acd = 111$ 或者 $acd = 011$ 时， $f = \bar{a}d + ac$ 都会产生 $f = 1$ 。然而，由于逻辑门存在延迟， f 在 0.2 ~ 0.3ns 内不会一直为 1，如时序图所示。

58

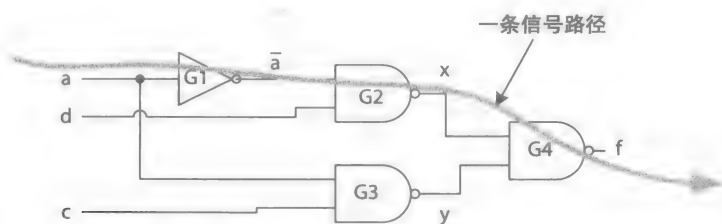


图 2-19 具有中间信号名字的 $f = \bar{a}d + ac$ 电路图； f 不依赖于 b

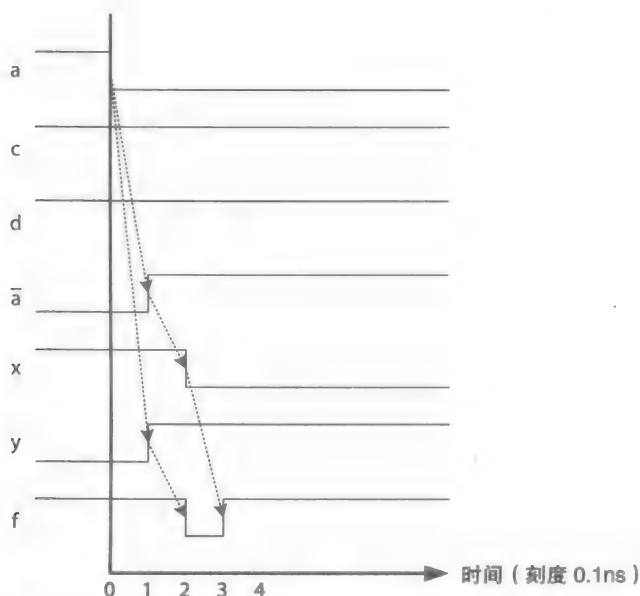


图 2-20 当 a 从 1 转变为 0 时，图 2-19 中电路的时序图

在时间步 1 中 a 信号在 G1 逻辑门中从初始值 0 到 1 的改变，使得 0.1ns 的延迟后 \bar{a} 从 0 变为 1。当 a 变成 0 即 \bar{a} 变为 1 时，在图中用箭头表示这种变化。同时，在时间步 1 中，信号 y 在 G3 逻辑门中在 0.1ns 的延迟后从 0 变成 1。信号 \bar{a} 在 0.1ns 时的改变，引起了在逻辑

门 G2 0.1ns 的延迟之后，信号 x 在时间步 2 中从 1 到 0 的改变。当 \bar{a} 变为 1， x 变为 0 时，在图中用箭头表示这种变化。

59

在时间 = 0.1ns 时，信号 x 和 y 都为逻辑 1，在逻辑门 G4 0.1ns 延迟后， f 从 1 转变为 0。在时间 = 0.2ns 时，信号 x 转变成 0，时间 = 0.3ns 时， f 在 0.1ns 后转变为 1。此时 f 一直保持为 1。

信号 f 这种非预期的从 1 变成 0 又变为 1 的变化称为冒险或者电子脉冲，它取决于电路中逻辑门和线路的延迟（此处线路延迟忽略不计）。在这个例子中，当函数 $f = \bar{a}d + ac$ 输入从 $acd = 111$ 变为 $acd = 011$ 时，称其有 1- 冒险。当一个与或（或者只有或非门）电路的时序图展示了一个从 0 到 1 又回到 0 的非预期的变化，我们就说这个电路有 0- 冒险。

冒险违背了组合电路的预期行为，且必须要阻止其影响数字系统的状态（例如寄存器内容）。在第 4 章中，我们将介绍一种有固定期限的时钟信号，一直在 1 0 1 0 1 0 1 0... 中来回变换，以控制寄存器的加载时间。在图 2-20 中，输出 f 在电路输入做出改变之后的 0.3ns 是合法的。时钟周期取决于信号的传播延迟和其他在第 4 章和第 5 章中提到的延迟。

2.6.1 信号传播延迟

通常，在电路中有多条从输入到一个或多个输出的信号路径。例如，图 2-19 中的输出信号 f 由信号路径 G1-G2-G4 或者 G3-G4 决定。当输入信号改变，传播到输出信号改变所需的时间是由信号路径上的逻辑门的数目和大小及线路延迟决定的。最长路径所产生的延迟称为电路的传输延迟。忽略线路延迟，电路的传输延迟与电路中最长信号路径的逻辑门数量成正比。图 2-19 中路径 G1-G2-G4 是最长路径。这样，电路传输延迟和三个逻辑门的延迟成正比，或者为 0.3ns，如图 2-20 所示；这里简单地假设每个逻辑门有 0.1ns 的延迟。0.3ns 的延迟也是消除输出 f 的 1- 冒险所需的时间，也是当其电路输入从 $acd = 111$ 到 $acd = 011$ 进行转换时，根据其逻辑表达式使得电路的输出 $f = 1$ 的所需时间。

通常来说，有多个输出电路模型的传输延迟是由从其输入到输出的最长路径决定的。在这个例子中，一些独立的输出可以有短一些的传输延迟；而然，在电路中至少有一个输出信号有最长的路径，这决定了这个电路模型的传输延迟。

实现 SOP 和 POS 表达式的电路通常有一些信号路径包含二级逻辑门或者三级逻辑门，包括初始的非门。这将导致传输延迟只与两个或者三个逻辑门延迟成正比。这样 SOP 或者 POS 表达式通常通过加快输出速度来提高性能。

然而，包含许多变量逻辑项的表达式可能由于逻辑门的扇入限制（稍后讨论）无法使用二级或者三级电路实现。在这个例子中，一个表达式必须可以分成更小的表达式，每一个表达式用需要更少扇入的逻辑门的更小的电路实现。然后更小的电路连接起来，构成最终的多级电路，不考虑最初的非门。例如，FPGA，作为一个可编程芯片，通常只有有限的资源而无法实现任意大小的 SOP 或者 POS 表达式。FPGA 通常比高性能的定制芯片要慢。

60

2.6.2 扇入和扇出

扇入是指一个逻辑门可以拥有的输入个数，扇出是指一个逻辑门可连接输出的连接数。例如，图 2-21 展示了一个有 3 个扇入和 5 个扇出的与非门。一个非门的扇入永远为 1。一个异或门和异或非门的扇入通常为 2。与门、或门、与非门和或非门的扇入可以为多个；然而，每一个门为了运行正常，都拥有最大扇入和最小扇出限制（例如 8）。

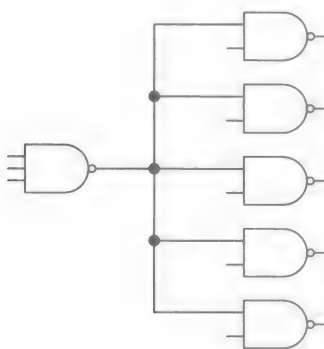


图 2-21 有 3 个扇入和 5 个扇出的与非门

2.7 其他逻辑门

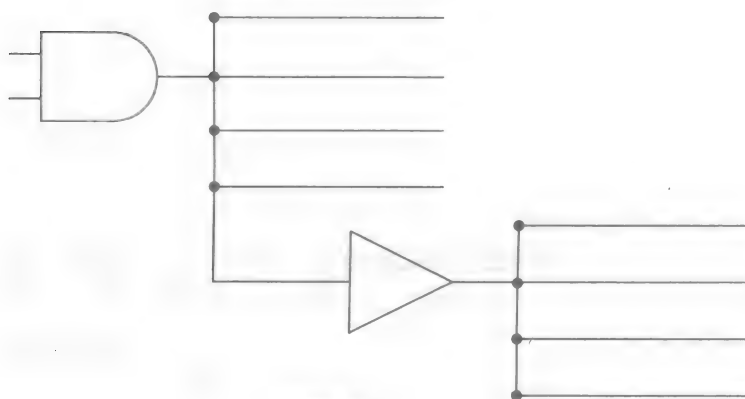
作为对之前标准逻辑门的补充，这节里将介绍一些其他对设计数字系统来说必需的其他逻辑门。这些逻辑门有缓存、集电极开路（OC）缓冲区和三态缓存。

2.7.1 缓存

缓存的符号和真值表如图 2-22a 所示。一个缓存不会改变其输入信号；只是简单地放大它。假设与门的扇出为 5，缓存可以增加与门的扇出至 5 ~ 9，如图 2-22b 所示。



a) 缓冲门及其真值表



b) 增加扇出

图 2-22 缓存门：a) 缓冲符号及其真值表；b) 用于增加扇出的缓存

2.7.2 集电极开路缓冲区

OC 缓冲区和缓存门类似，除了当其输入为逻辑 1 时，其输出变为高阻抗，如 Z (图 2-23a) 所示。高阻抗信号既不是逻辑 0 也不是逻辑 1，而是显示为电隔离，就像导线是“浮动”的

而不是连接的。图 2-23b 展示了有两个 OC 缓冲区的电路。每一个逻辑门的输出不是 0 就是 Z，这样输出可以连接到一起产生一个输出信号 f 。Z 输出可以使用连接到其他电源（例如 5.0V）或者电源接地（0.0V）的电阻“拉起”或者“拉下”变为逻辑 1 或者逻辑 0。图 2-23b 中的 Z 输出是拉起。

图 2-24 展示了使用图 2-23b 中有两个输入 a 和 b 的电路的高阻抗输出行为。当 $a = 0$ 和 $b = 0$ 时，OC 缓冲区 B1 和 B2 输出 0，这样就将 f 连接到地，即逻辑 0（图 2-24a）。当 $a = 1$ 和 $b = 1$ 时，OC 缓冲区 B1 和 B2 输出为 Z（浮动的）；这就让 f 连接到电源，即为逻辑 1（图 2-24d）。当 $a = 0$ 和 $b = 1$ 或者 $a = 1$ 和 $b = 0$ 时，其中一个缓存输出 0 而其他缓存输出 Z，这样就把 f 和逻辑 0 相连，如图 2-24b 和图 2-24c 所示。4 个例子总结了一张如图 2-23b 的真值表。真值表展示了一个与逻辑，和这个例子中的叫作线 - 与逻辑的电路。线逻辑电路可以拥有很大的扇入。

线 - 与和线 - 或逻辑是两种常见的线逻辑电路。例如，线逻辑电路可以用于设计有扩展槽的计算机系统。在这个例子中，线逻辑电路可以在计算机系统中通过在一个叫作设备控制器接口（DCI）的计算机扩展槽中插入扩展卡来增加功能，例如一个有 n 个输入的线 - 或逻辑电路，可以对 n 个信号进行与操作，信号可以来自不同的设备，如图 2-23c 所示。这个设备接口将在第 9 章中更详细地进行讨论。

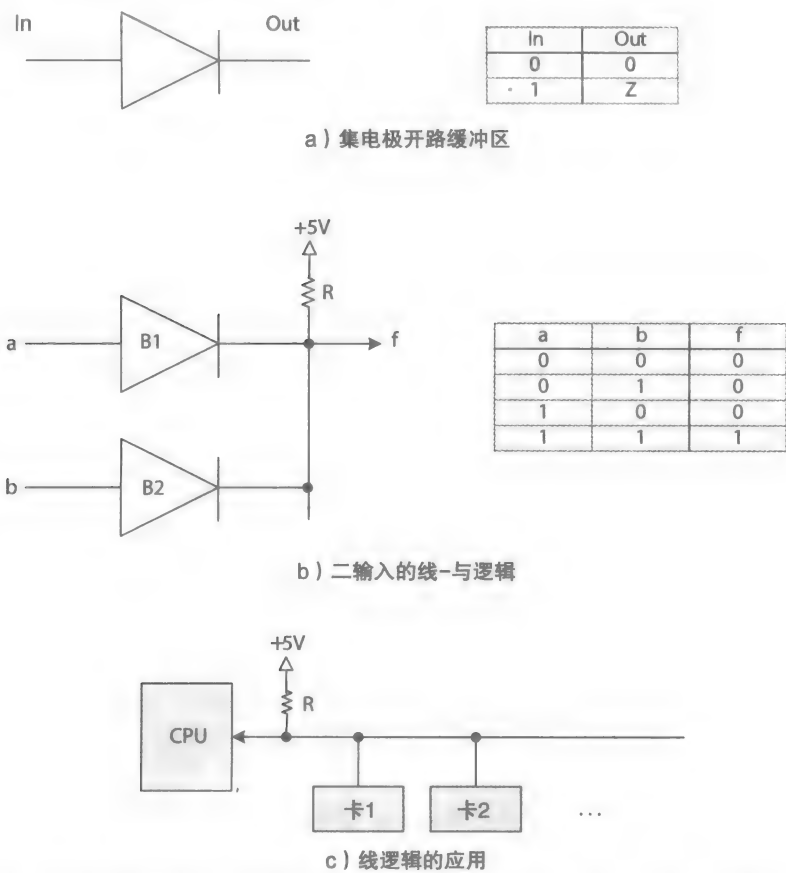


图 2-23 集电极开路缓冲区及其应用：a) 符号和真值表；b) 一个二输入的线 - 与逻辑和真值表；c) 扩展槽设计图

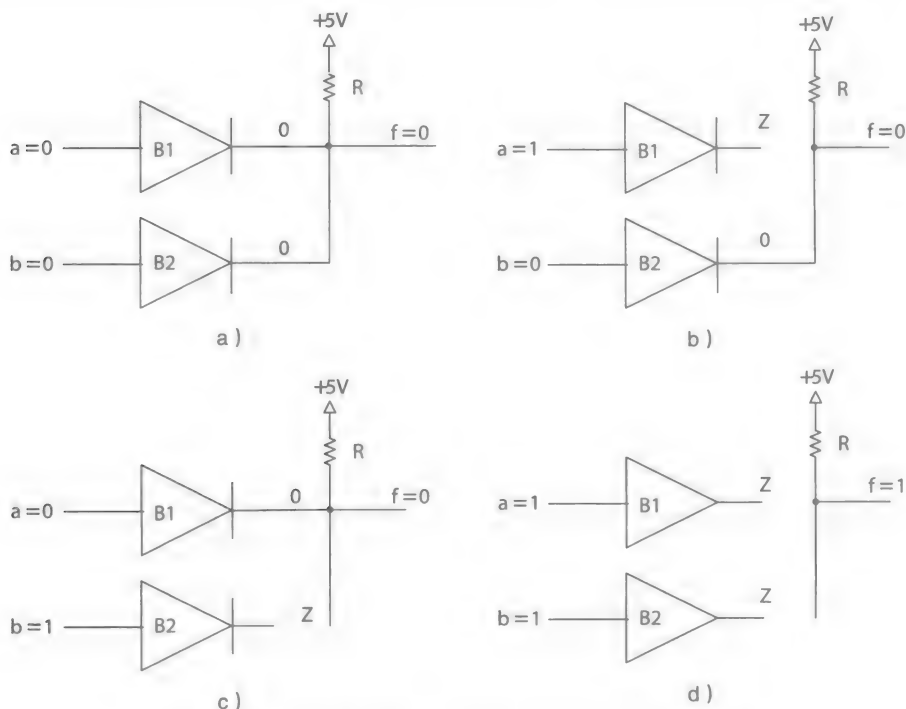


图 2-24 二输入的线-与逻辑 4 种不同的输入情况

2.7.3 三态缓存

图 2-25 展示了三态缓存及其真值表。三态缓存是一个缓存和一个 OC 缓存的组合。当使能时 ($e = 1$)，它像一个缓存一样工作，但当禁止时 ($e = 0$)，其输出变为 Z 。三态缓存用于当输入为两个或多个信号需要共享一条叫作总线的线路的情况中。通常，总线可以用于很多条线路。图 2-26 展示了三态缓存到 1 位总线的连接。每一次一个使能信号 e_1 、 e_2 或者 e_3 可以使得对应的信号 a 、 b 或 c 放到总线上。其他禁用的三态缓存将输出 Z (“浮动的”)，这样可以使其与总线隔离。



| e | In | Out |
|---|----|-----|
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

图 2-25 三态缓存及其真值表

如果一个电路模型输出到总线和从总线中输入，总线连接可以是双向的。图 2-27a 展示了一个双向总线连接的例子，使用一个缓存从总线中输入和一个三态缓存输出到总线中。在双向总线中传输的数据项有一个源模块和一个目的模块。源模块用一个三态缓存在总线中放置一个数据项，目的模块用一个缓存输入数据。如果总线扇出在目的模块中大于 1，缓存可以保护系统在源模块中免受扇出干扰；这就是，一个总线信号在目的模块中与两个或者多个逻辑门相连，如图 2-27a 中的目的模块 C。

总线降低了互连的开销。其可以代替多个在多个模块中一对一的连接 (图 2-27b)，只要数据 (位数) 通过总线发送的速度 (频率) 可以足够大来处理负载。这个总线数据速度被称为带宽。例如，在图 2-27a 中，每 10ns 一位数据可以在一位总线上传输，如果使能信号 e_1 、

e2 和 e3 每 10ns 只有一个有效，或者其中一个使能信号在多个 10ns 中有效。可以在每 10ns 中传送 1 位数据的 1 位总线的带宽与可以在每 100ns 中可以传送 10 位数据的 10 位总线的带宽是一样的；在 100ns 中，两条总线都可以传送 10 位数据。这样，总线的宽度（线路的数目）和总线的速度（频率）决定了总线的带宽。

64

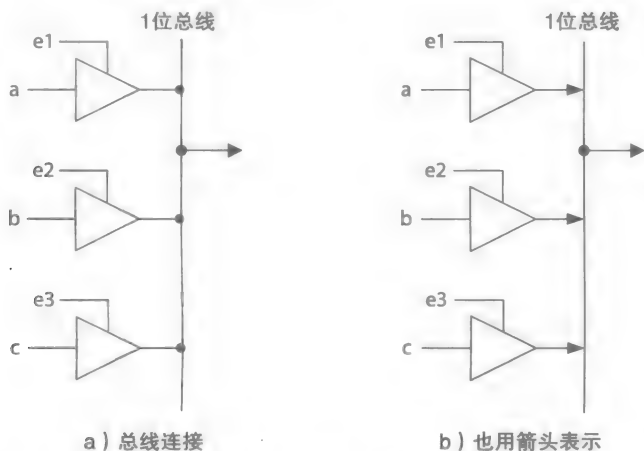


图 2-26 共享 1 位总线的三态缓存：a) 实际连接；b) 连接通常用箭头表示

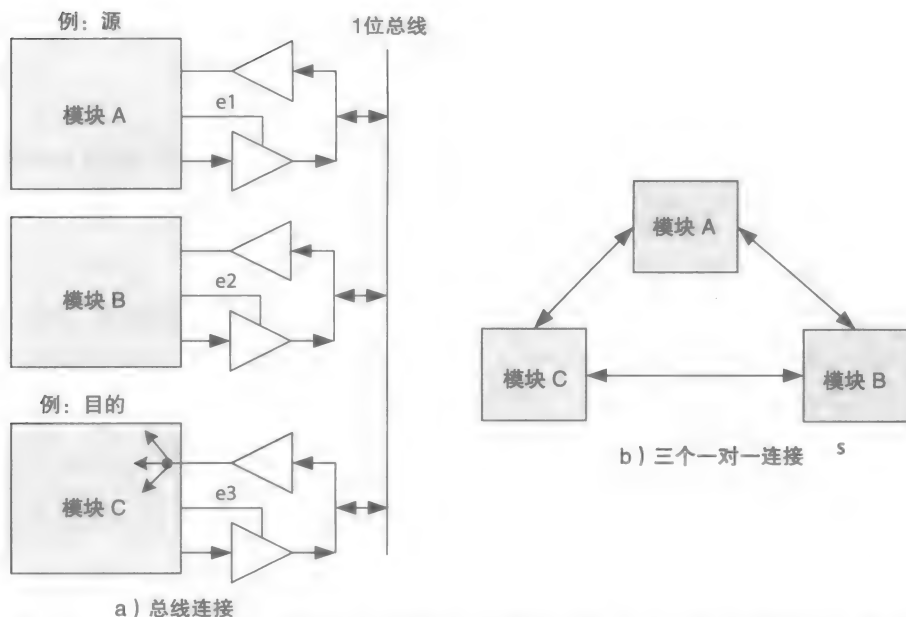


图 2-27 模块互连：a) 使用同一总线的三个模块互连；b) 一对一连接的三个模块

65

两个三态缓存可以用于收发器（传输 / 接收）电路连接，例如，如图 2-28 所示的两条分开的总线。每一个收发器电路都有两条总线之间的双向连接。方向（dir）信号决定了数据的方向，从总线 A 到总线 B 或者从总线 B 到总线 A。当使能（e）信号有效时，数据连接到两条总线上，并保持它们一直连接。

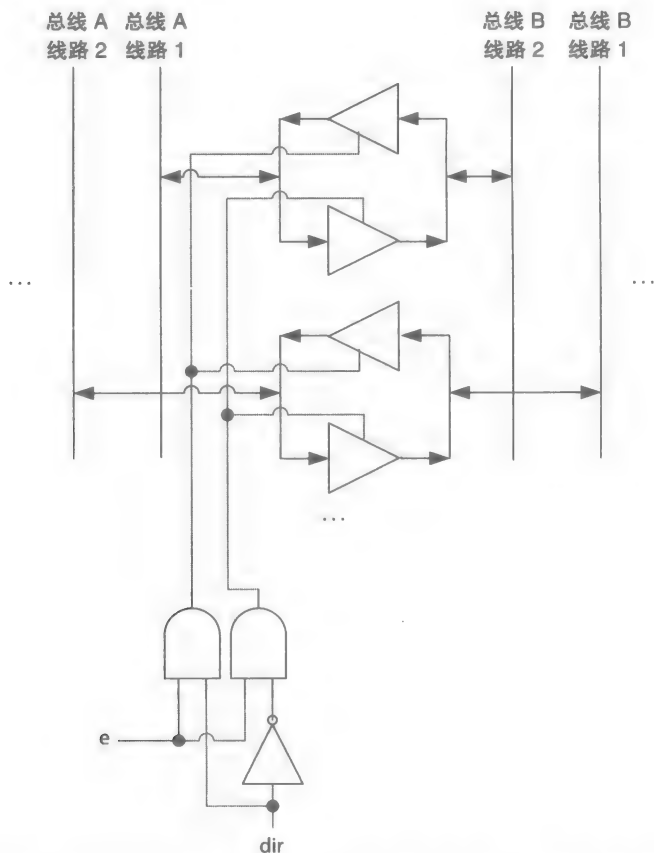


图 2-28 显示两条总线的收发器模块 [2]; dir 信号表示数据方向, e 信号表示两条总线的连接

2.8 设计实例

在第 1 章中讨论过, 一个数据通路包括许多电路模块。本节我们将讨论一些常见的但是比较小型的组合电路模块。在第 1 章中图 1-1 中所示的“选择器”模块也被叫作多路转换器。其他将讨论的实例包括简单的加法器、译码器和编码器模块。

1 位加法器, 也叫作全加器 (FA), 产生两个 1 位输入的和, 并输出一个 0 或 1 的传位进位。译码模块将数字 A (0、1、2 等) 转换成对应的输出信号 (例如, f_0 、 f_1 、 f_2 等)。任何时候都只有一个输出 f_0 、 f_1 等是有效的。另一方面, 编码器所做的工作是译码器的逆向工作, 将产生与输入信号对应的数字。这节中我们将讨论下列实例:

- 高电平信号 FA 设计实例。
- 1 位 2-1 的多路选择器和 1 位 4-1 的多路选择器设计实例。
- 低电平输出信号的 1-2 译码器设计实例。
- 低电平输入信号的 3-2 编码器设计实例。

2.8.1 全加器

一个全加器有三个 1 位输入, 其中一个输入是进位输入 (c_{in}), 输出为一位和 (s) 和一

位进位输出 (c_{out}), 如图 2-29 所示。表 2-8 展示了全加器的真值表。在表中的每一种情况中, s 和 c_{out} 都由三位数据 a 、 b 和 c_{in} 的和决定。多全加器模块可以用于设计大型加法器, 这将在下一章中讨论。

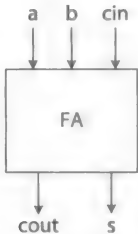


图 2-29 FA 的模型图

表 2-8 FA 的真值表

| a | b | c_{in} | c_{out} | s |
|-----|-----|----------|-----------|-----|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

67

s 和 c_{out} 的最小 SOP 表达式如下求得：

$$\begin{aligned} s(a, b, c_{in}) &= \Sigma(1, 2, 4, 7) \\ c_{out}(a, b, c_{in}) &= \Sigma(3, 5, 6, 7) \\ s &= \overline{a}bc_{in} + \overline{a}b\overline{c_{in}} + a\overline{b}\overline{c_{in}} + abc_{in} \\ c_{out} &= ab + ac_{in} + bc_{in} \end{aligned}$$

| bc_{in} : | 00 | 01 | 11 | 10 |
|-------------|----|----|----|----|
| $a: 0$ | | 1 | | 1 |
| 1 | 1 | | 1 | |

| bc_{in} : | 00 | 01 | 11 | 10 |
|-------------|----|----|----|----|
| $a: 0$ | | | 1 | |
| 1 | | 1 | 1 | 1 |

(2-8)

作为一种选择, s 和 c_{out} 表达式也可以用异或门写成如公式 (2-9) 所示, 这样就简化了电路的门级结构, 如图 2-30 所示。然而, 这个解法将导致比公式 (2-8) 所实现电路的更长的传输延迟。

$$\begin{aligned} s &= \overline{a}bc_{in} + \overline{a}b\overline{c_{in}} + a\overline{b}\overline{c_{in}} + abc_{in} \quad (\text{规范 SOP}) \\ &= c_{in}(\overline{a}b + ab) + \overline{c_{in}}(\overline{a}b + a\overline{b}) \\ &= c_{in}(a \oplus b) + \overline{c_{in}}(a \oplus b) \\ &= a \oplus b \oplus c_{in} \\ c_{out} &= \overline{a}bc_{in} + \overline{a}b\overline{c_{in}} + ab\overline{c_{in}} + abc_{in} \quad (\text{规范 SOP}) \\ &= (\overline{a}b + ab)c_{in} + ab(\overline{c_{in}} + c_{in}) \\ &= (a \oplus b)c_{in} + ab \end{aligned} \tag{2-9}$$

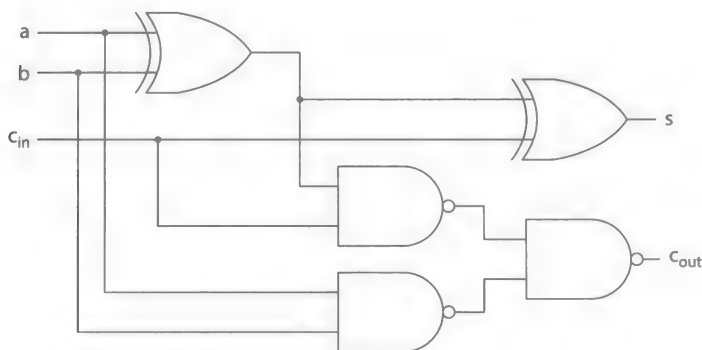


图 2-30 FA 电路的另一种表达

传输延迟估计

假设与非门有 0.1ns 的延迟，公式 (2-10) 估算了公式 (2-8) 中信号 s 和 c_{out} 的传输延迟，用符号 Δ 表示。信号 s 和 c_{out} 的 SOP 表达式分别有三级和二级的逻辑门。在信号 s 和 c_{out} 的延迟估算中我们忽略掉了线路延迟。

$$\begin{aligned}
 \Delta s &= \Delta_{\text{not}} + \Delta_{\text{nand}} + \Delta_{\text{nand}} \\
 &= 0.1\text{ns} + 0.1\text{ns} + 0.1\text{ns} \\
 &= 0.3\text{ns} \\
 \Delta c_{\text{out}} &= \Delta_{\text{nand}} + \Delta_{\text{nand}} \\
 &= 0.1\text{ns} + 0.1\text{ns} \\
 &= 0.2\text{ns}
 \end{aligned} \tag{2-10}$$

公式 (2-11) 展示了公式 (2-9) 中信号 s 和 c_{out} 的延迟估算，SOP 表达式中的异或门有 0.3ns 的延迟：

$$\begin{aligned}
 \Delta s &= 2 * \Delta_{\text{xor}} \\
 &= 2 * 0.3\text{ns} \\
 &= 0.6\text{ns} \\
 \Delta c_{\text{out}} &= \Delta_{\text{xor}} + \Delta_{\text{nand}} + \Delta_{\text{nand}} \\
 &= 0.3\text{ns} + 0.1\text{ns} + 0.1\text{ns} \\
 &= 0.5\text{ns}
 \end{aligned} \tag{2-11}$$

2.8.2 多路选择器

1 位 2-1 的多路选择器，或者缩写 MUX，是如图 2-31 所示的简单组合电路。输入 x 和 y 都为 1 位数据，信号 s (选择器信号) 导致 MUX 输出 x 或者 y 。如电路框图所示，标记 1 和 0 分别用来标记 x 和 y ，反映在 MUX 的真值表里 (表 2-9)。当 $s = 0$ 时，MUX 输出 y ，当 $s = 1$ 时，MUX 输出 x 。其最小 SOP 表达式由以下式子决定：

$$\begin{aligned}
 r(s, x, y) &= \sum(1, 3, 6, 7) \\
 r &= \bar{s}y + sx
 \end{aligned}$$

| xy: | 00 | 01 | 11 | 10 |
|------|----|----|----|----|
| s: 0 | | 1 | 1 | |
| 1 | | | 1 | 1 |

(2-12)

例如，当 $s = 0$ 时，公式 (2-12) 计算结果符合预期，即 $r = y$ ，如下所示：

$$\begin{aligned}
 r &= \bar{0}y + 0x \\
 &= 1y + 0 \\
 &= y + 0 \\
 &= y
 \end{aligned}$$

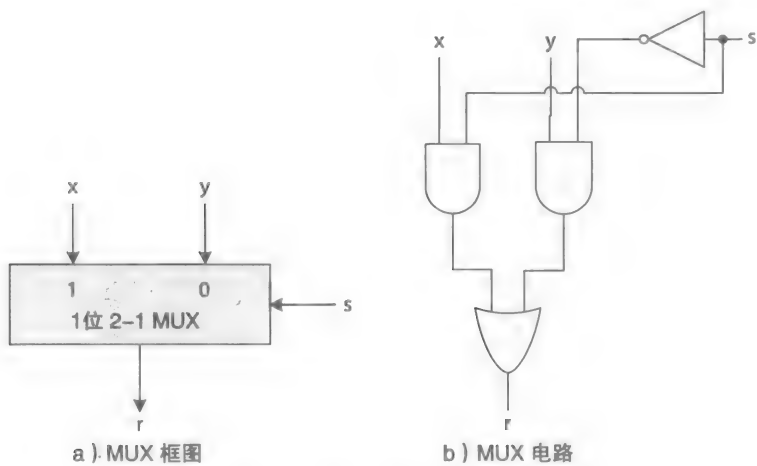


图 2-31 1 位 2-1 MUX 的电路框图

表 2-9 1 位 2-1 的 MUX 真值表

| s | x | y | r |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

类似地，当 $s = 1$ 时，MUX 输出（或者选择）输入 x 。图 2-32 展示了拥有 4 位输入 w 、 x 、 y 和 z 的 1 位 4-1 的 MUX 电路框图，输入分别标记为 3 到 0 的数字。MUX 需要两个选择信号，标记为 s_1 和 s_0 。表 2-10 展示了其简单的真值表。其扩展真值表有 6 个输入，比我们使用 K 图所允许的 4 个输入要大。除了使用 K 图，有两种方法来确定 4-1 MUX 的 SOP 表达式：1) 用 Espresso 软件；2) 对有 4 个输入和 2 个选择信号的公式 (2-12) 进行推算。这就是当 $s_1s_0 = 0 = (00)_2$ 时，MUX 将输出 z ，当 $s_1s_0 = 1 = (01)_2$ 时，输出为 y ；当 $s_1s_0 = 2 = (10)_2$ 时，输出为 x ，当 $s_1s_0 = 3 = (11)_2$ 时，输出为 w 。其最小 SOP 表达式为：

$$r = \overline{s_1}\overline{s_0}z + \overline{s_1}s_0y + s_1\overline{s_0}x + s_1s_0w \tag{2-13}$$

例如，当 $s_1s_0 = 2 = (10)_2$ 时，公式 (2-13) 计算结果符合预期，为 $r = x$ ，如下所示：

$$\begin{aligned} r &= \overline{1}\cdot\overline{0}\cdot z + \overline{1}\cdot 0\cdot y + 1\cdot\overline{0}\cdot x + 1\cdot 0\cdot w \\ &= 0\cdot 1\cdot z + 0\cdot 0\cdot y + 1\cdot 1\cdot x + 1\cdot 0\cdot w \\ &= 0 + 0 + x + 0 \\ &= x \end{aligned}$$

4-1 MUX 的电路框图如图 2-33 所示。当 MUX 的大小增大，其扇入和扇出需求也相应地增大。考虑上述的 2-1 和 4-1 MUX。它们最大扇入和扇出需求分别为 2 和 2 以及 4 和 3。大型 MUX，如果使用现在讨论的方法去设计，那么将导致扇入和扇出问题。在第 3 章中，我们将讨论大型组合电路的设计方法，首先将大型设计问题分为多个小型问题，然后对于每

一个小型问题，使用这一章学到的方法去解决。之后小型电路将组合在一起形成大型组合电路，其可以避免任何的扇入和扇出问题。

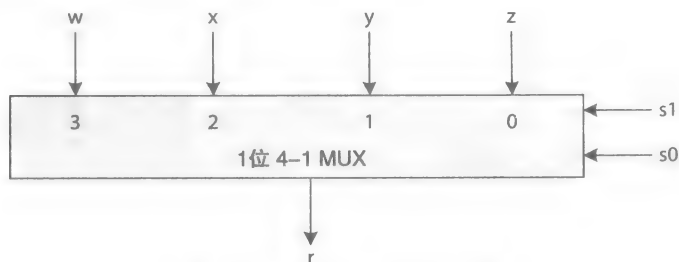


图 2-32 1 位 4-1 MUX 框图

表 2-10 1 位 4-1 MUX 的化简真值表

| s_1 | s_0 | r |
|-------|-------|-----|
| 0 | 0 | z |
| 0 | 1 | y |
| 1 | 0 | x |
| 1 | 1 | w |

71

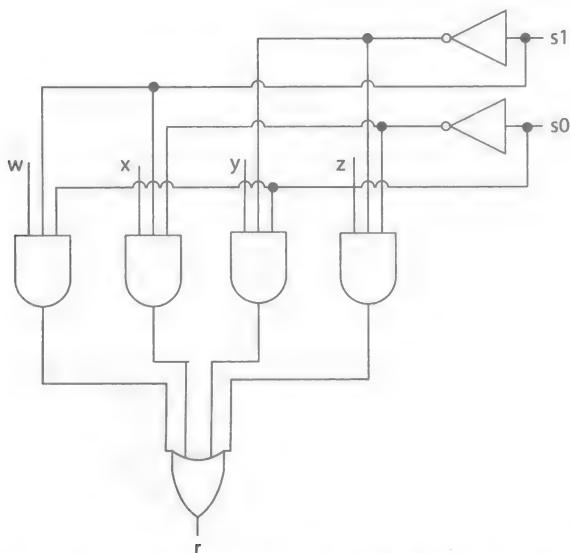


图 2-33 1 位 4-1 MUX 的电路框图；最大扇入 = 4，最大扇出 = 2

2.8.3 译码器

低电平输出的 1-2 译码器电路框图如图 2-34 所示。低电平标记引脚通常在基础引脚上加一个气泡表示，如图 2-34b 所示。然而，图 2-34b 中的气泡不仅仅是简单的非门表示。表 2-11 展示了译码器的真值表。只有一个或者没有 $_f_1$ 和 $_f_0$ 输出有效，这依赖于信号 v 和 e 的值。当 $e = 1$ 且 $v = 0$ 时， $_f_0 = 0$ (有效)。当 $e = 1$ 且 $v = 1$ 时， $_f_1 = 0$ (有效)。否则，当

$e = 0$ (不活跃) 时, $_f1$ 和 $_f0$ 都为 1 (无效)。

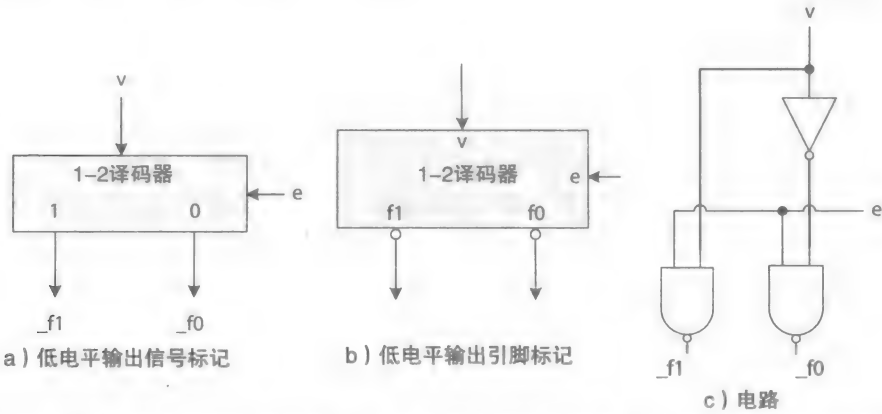


图 2-34 1-2 译码器的电路框图：a) 有信号名的电路框图；b) 有引脚标记的信号框图；c) 译码器电路

72

表 2-11 低电平输出信号的 1-2 译码器真值表

| e | v | $_f1$ | $_f0$ |
|-----|-----|--------|--------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

信号 $_f1$ 和 $_f0$ 的表达式可以表达为 SOP 或者 POS。然而在这个例子中，对于这些信号的 SOP 或者 POS 表达式都是一样的。只有与非门译码器电路只有一个逻辑门的延迟，不包括非门 (图 2-34c)。

$$_f1(e, v) = \Sigma(0, 1, 2) = \Pi(3)$$
$$_f1 = \bar{e} + \bar{v} + e\bar{v}$$

$$_f0(e, v) = \Sigma(0, 1, 3) = \Pi(2)$$
$$_f0 = \bar{e} + v = e\bar{v}$$

| | | |
|--------|-----|-----|
| v : | 0 | 1 |
| $e: 0$ | (1) | (1) |
| 1 | (1) | |

| | | |
|--------|---|-----|
| v : | 0 | 1 |
| $e: 0$ | | |
| 1 | | (0) |

| | | |
|--------|-----|-----|
| v : | 0 | 1 |
| $e: 0$ | (1) | (1) |
| 1 | | (1) |

| | | |
|--------|-----|---|
| v : | 0 | 1 |
| $e: 0$ | | |
| 1 | (0) | |

大型译码器也可以运用不同的方法来设计避免扇入和扇出的问题。译码器有许多应用，可以用于设计例如一些内存或者 CPU 数据通路。译码器可以用于解释内存地址，这样地址的内容可以进行读写操作。当需要写入一个寄存器文件时 (第 1 章)，译码器也可以用于译码一个寄存器数值。

2.8.4 编码器

低电平输入信号的 3-2 编码器的电路框图如图 2-35a 所示，且低电平标记引脚也被标记为一个小圆圈，如图 2-35b 所示。表 2-12 展示了编码器的真值表。在表中输入信号用 0 ~ 2

的数字来表示。编码器电路输出一个由活跃的输入定义的数字。例如，当 $\underline{z} = 0$ （活跃的）， $\underline{y} = 1$ （不活跃）且 $\underline{x} = 1$ （不活跃），编码器输出 $r_1r_0 = (00)_2 = 0$ ，正确识别有效信号 \underline{z} 为数字 0。然而，当编码器的输入没有一个是活跃的时候，定义另外一个输出信号 a （输入活跃）是有必要的。当 $a = 1$ （有效）时，表示一个或者多个信号 \underline{x} 、 \underline{y} 和 \underline{z} 为活跃的，这样 2 位输出 r_1r_0 被定义为活跃的信号。另一方面，当 $a = 0$ （无效）时，输出 $r_1r_0 = (00)_2$ 被忽略。

两个或两个以上的编码器输入在同一时间变为活跃是有可能的。例如，当 $\underline{x} = 0$ ， $\underline{y} = 0$ 和 $\underline{z} = 1$ 时，编码器必须遵照某一信号优先级输出信号 \underline{x} 或者信号 \underline{y} 对应的数字。这样的编码器叫作**优先级编码器**。

73

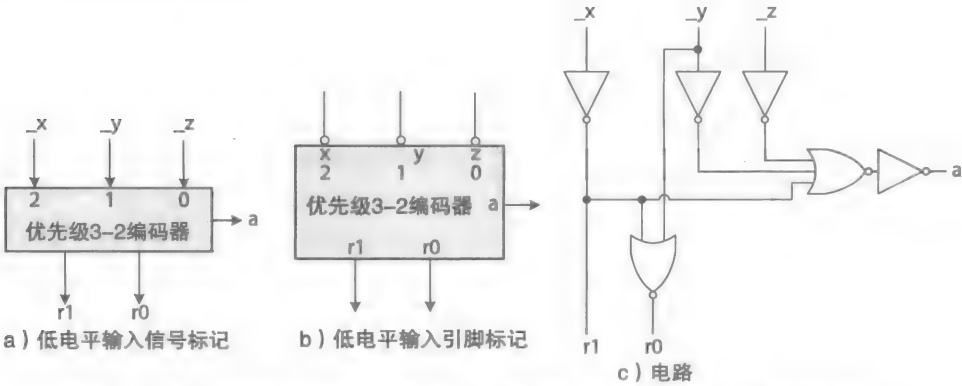


图 2-35 3-2 编码器的电路框图：a) 有信号名的电路框图；b) 有引脚标记的电路框图；c) 编码器电路

表 2-12 展示了 3-2 优先级编码器的真值表（图 2-35），输入信号 \underline{x} 有最高优先级而 \underline{z} 的优先级最低。当 $\underline{x} = 0$ （活跃）、 $\underline{y} = 0$ （活跃）和 $\underline{z} = 1$ （非活跃）时，编码器输出 $a = 1$ 和 $r_1r_0 = (10)_2$ ，标记 \underline{x} 为最高优先级的输入信号。

信号 a ， r_1 和 r_0 的 POS 表达式如下所示：

$$r_1(x,y,z) = \Pi(4,5,6) + \Pi_d(7)$$
$$r_1(x,y,z) = \underline{\bar{x}}$$
$$r_0(x,y,z) = \Pi(0,1,2,3,6) + \Pi_d(7)$$
$$r_0(x,y,z) = (\underline{\bar{x}})(\underline{\bar{y}}) = \underline{\bar{x}} + \underline{\bar{y}}$$
$$a(x,y,z) = \Pi(7)$$
$$a(x,y,z) = \underline{\bar{x}} + \underline{\bar{y}} + \underline{\bar{z}} = \underline{\bar{x}} + \underline{\bar{y}} + \underline{\bar{z}}$$

| | | | | |
|---------------------------------|----|----|----|----|
| $\underline{y} \underline{z}$: | 00 | 01 | 11 | 10 |
| $\underline{x}: 0$ | | | | |
| 1 | 0 | 0 | d | 0 |

| | | | | |
|---------------------------------|----|----|----|----|
| $\underline{y} \underline{z}$: | 00 | 01 | 11 | 10 |
| $\underline{x}: 0$ | 0 | 0 | 0 | 0 |
| 1 | | | d | 0 |

| | | | | |
|---------------------------------|----|----|----|----|
| $\underline{y} \underline{z}$: | 00 | 01 | 11 | 10 |
| $\underline{x}: 0$ | | | | |
| 1 | | | 0 | |

表 2-12 低电平输入信号 3-2 编码器的真值表

| | | | | | |
|-----------------|-----------------|-----------------|-----|-------|-------|
| \underline{x} | \underline{y} | \underline{z} | a | r_1 | r_0 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |

(续)

| $_x$ | $_y$ | $_z$ | a | r_1 | r_0 |
|-------|-------|-------|-----|-------|-------|
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | d | d |

74

编码器也可以设计为没有输出信号 a ，如图 2-36 所示。它被设计成 4-2 的编码器形式，没有信号 a ，但是输入数字 0 是未使用的，且与电源连接（或者高电平有效的输入接地），这样可以有效地改变为 3-2 编码器。当输入信号 $_x$ 、 $_y$ 和 $_z$ 中没有一个是活跃的时候，编码器的输出为 $r_1r_0 = (00)_2$ （原书有误——译者注），表示输入是不活跃的。当一个或多个输入信号变为活跃时，编码器分别输出 3、2 和 1 来定义 $_x$ 、 $_y$ 和 $_z$ 为活跃信号。这样消除了产生信号 a 相应的逻辑，并在设计时减少了电路信号，如图 2-35 所示。连接到电源的输入可以在内部实现。再次地，在设计一个大型编码器时，将用不同的方法来避免扇入扇出的问题。

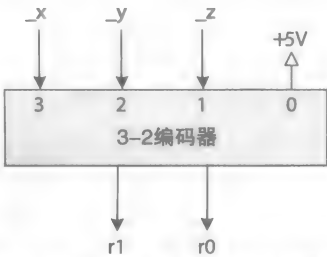


图 2-36 没有输入活跃的输出的 3-2 编码器框图

编码器也可以拥有许多应用，特别是在设计母版中。例如，当外部信号变为有效时，编码器可以快速地通知 CPU。有效外部信号可以由一个输入 / 输出设备产生或者由板上的给 CPU 提供服务的模块产生。

2.9 实现

现代数字电路设计师依靠 CAD 工具来将设计模型翻译成实现数据。数字设计 CAD 工具将数字电路的描述综合（翻译）成优化过和对技术依赖的门级描述，称为网表。特定用途的集成芯片（ASIC）和 FPGA 都是非定制 IC 技术的例子。处理器芯片是典型的定制 IC 技术的例子。电路可以用框图表示，也可以用 HDL 语言描述，或者同时使用这两种方法描述。然而，现代 CAD 工具需要将电路描述成 HDL 语言。

2.9.1 可编程逻辑器件

可编程逻辑器件（PLD）是预制的，即不包含任何制造缺陷的封闭现成设备。它们可以被编程（例如配置）去实现网表，这个过程是立即实现的，有时甚至是动态的过程。简单可编程逻辑器件（SPLD）是所有可编程逻辑器件中最简单的一种。SPLD 使用线逻辑来实现逻辑表达式，且通常用于实现小型数字电路。复杂可编程逻辑器件（CPLD）是通过芯片内部配置线通道来实现更复杂逻辑电路的下一代可编程逻辑器件。

75

我们在第 1 章中简要地讨论过，FPGA 是一种包含若干可配置逻辑块（CLB）、可配置线通道和与芯片（I/O）引脚相连的可配置 IO 模块 PLD 的现代版本。FPGA 可以看成是现代与 TTL 7400 芯片系列和电路板的等价。7400 系列是第一代被用于一般用途的 IC 芯片。该系列包含了标准逻辑门和更大的一些组合逻辑模块，例如 MUX、译码器和加法器，和一些用于设计时序电路的模块。在 20 世纪六七十年代，它们被用来设计小型和大型计算机。如今

7400 系列芯片有时会用于教学用途，特别是用在逻辑设计的引导课程中。

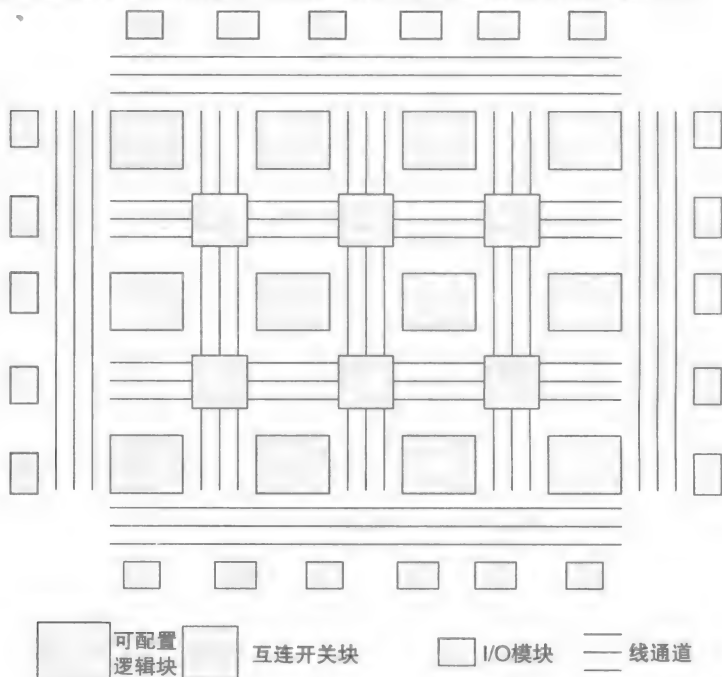


图 2-37 简单 FPGA 框图结构

FPGA 需要编程数据来配置和将用网表表示的 CLB 和 I/O 模块互连起来。一些 I/O 模块可配置成输入引脚和输出引脚。图 2-37 展示了包含 9 个可以实现一个或两个简单逻辑功能 CLB 的简单 FPGA 的内部构成。线通道和开关组用于每一个 CLB 输入和输出与其他 CLB 的互连，同时也将 I/O 模块与 I/O 引脚互连起来。

[76] 一些 FPGA 包含了配置存储器以根据需实现不同的网表。现代 FPGA 通常包含上千个 CLB，一些 FPGA 甚至会包含存储模块。还有一些片上系统 (SoC) FPGA 芯片 [3-4] 可以包含例如 CPU 和数字信号处理器 (DSP) 这样复杂的模块。对于这些芯片，我们可以更容易地不需要加工地设计定制和复杂的数字电路。Altera 和 Xilinx 公司提供了带有通用串行总线 (USB) 接口的 FPGA 开发组件 [3-5]。

2.9.2 设计流程

图 2-38 展示了数字电路的一般设计流程。设计流程一般包括设计输入、综合和实现阶段。设计流程里的每一个步骤会对目标电路产生一种不同的描述，其中每一个描述都会去验证设计是否有错误。

1. 设计输入

在设计这个步骤中，目标数字电路将被手工地用图形设计工具、HDL 或者两者共同去描述。在工业领域中，图形设计工具的严格使用许多年以来已逐渐减少，这有利于我们使用 HDL。我们将会 在 2.10 节讨论 Verilog HDL。图 2-39 展示了用图形设计和一种叫 [77] LogicWorks 的仿真工具来设计全加器的过程 [7]。LogicWorks 中没有合成工具。图形设计工具通常包含逻辑门库和一些常用的组合逻辑和时序逻辑电路模块。它也可以包含 7400 芯片

系列的库。此外，图形设计工具可以包括混合设计输入功能来进入数据通路原理图，数据通路中的模块都是从库中选择或者用 HDL 设计的。

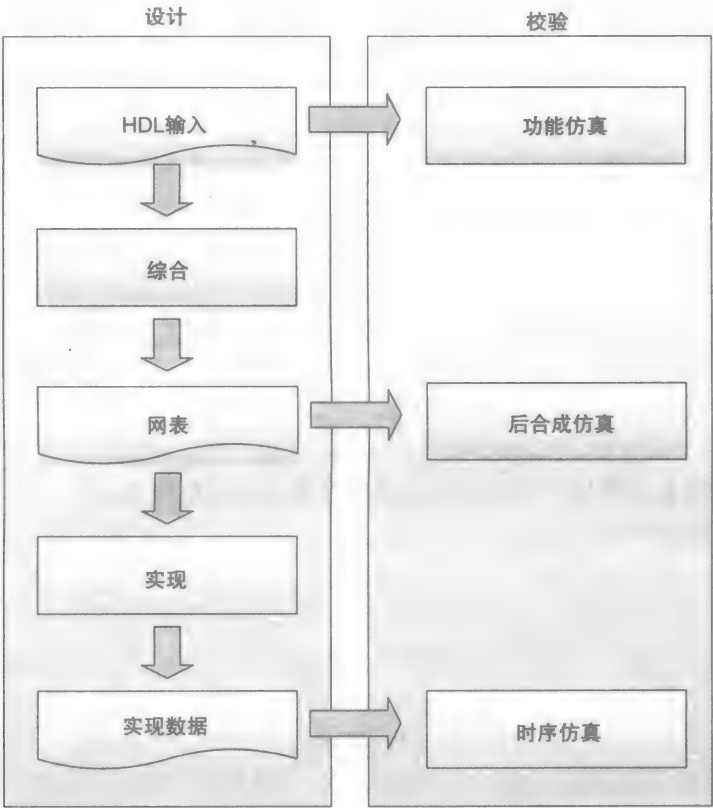


图 2-38 数字电路设计流程 [6]

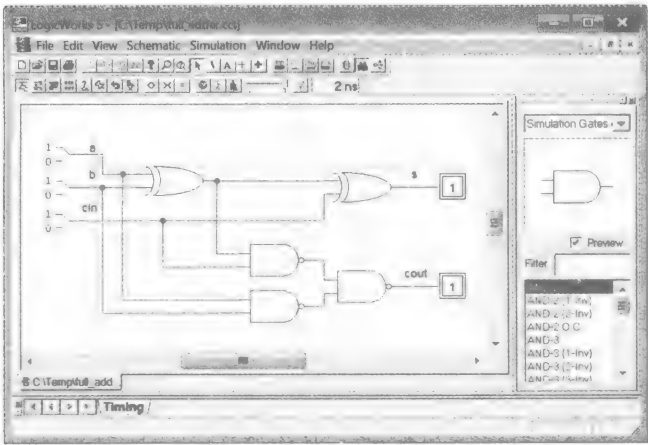


图 2-39 一种图形设计工具 LogicWorks 中全加器的电路图

2. 功能仿真

用图形和 / 或 HDL 来设计电路必须通过校验保证电路的运行符合预期。例如，给定输

入,如图 2-39 所示的全加器的输出能符合全加器的真值表吗?因为这个校验过程,特别是对于大型的电路会消耗许多时间,所以校验过程通常被分为功能校验过程、合成后校验过程和时序校验过程。功能仿真只用于校验设计的正确性,并不关心其实现中的问题。这是保证设计正确的第一步。如果电路的功能仿真没有错误,则合成就将作为电路实现的第一步;然而,设计还可能会有合成错误或者时序错误。

3. 合成后仿真

在合成步骤中,电路设计被解释为其对应的基于给定技术可用资源下的网表,例如给定的 FPGA 中每个 CLB 中可用的逻辑资源。每个 CLB 只能实现一些简单的逻辑表达式——例如,两个 4 变量的逻辑表达式。设计逻辑表达式需要把大的逻辑表达式分为较小的逻辑表达式——例如,不超过 4 变量的逻辑表达式。这个较小的逻辑表达式存储于电路网表中。

[78]

合成后仿真过程需要保证电路设计可以正确地翻译和其生成的网表必须正确地描述了目标电路。合成后仿真比时序仿真需要更少的处理时间。此外,一些延迟信息——例如,CLB 信号传输延迟,可能会在合成后仿真中出现。

4. 时序仿真

当网表映射到在计算机上模拟运行的目标设备中的可用资源之后,开始进行时序仿真。例如,用 FPGA 芯片的虚拟模型,网表通过一个叫布局布线的过程配置 CLB、I/O 模块和线通道。

在布局步骤中,网表中的最小表达式被分配到 CLB,电路初级输入和输出信号通过 I/O 模块被分配到 I/O 引脚。一些设计可能也需要复杂的模型——例如,已经存在在芯片上的 CPU、DSP 和存储器。

在布线的过程中,网表中的信号依赖信息通过在芯片上的线通道和开关组将不同 CLB 和 I/O 模块的信号连接起来。然而,布局和布线任务通常不是独立完成的;CLB 上的逻辑表达式分配、I/O 模块上的初始 I/O 信号都可以改变,这是为了:1) 最大化地利用芯片上的可用资源;2) 将传输延迟减到最小。时序仿真是用来保证设计时序需求。

2.10 硬件描述语言

Verilog 和 VHDL (VHSIC 或者高速集成电路)是两种用于描述数字电路的工业标准 HDL。HDL 用于在形式上描述数字电路,测试台用于生成电路测试(例如容器)。

如果电路被一组互连的模块进行描述,那么我们说这个 HDL 描述是结构的。这些模块可以很小,如与、或、与非等逻辑门,也可以大一些,例如译码器、多路选择器、加法器等。一些通常使用的大模块,如加法器,可以预定且在合成过程中使用。

如果 HDL 代码描述了模块中输入的关系且在描述输出时使用了例如“if-else”或者“case”的高级语言,那么我们称这个 HDL 描述为行为。

2.10.1 结构模型

本节中,我们将简要介绍 Verilog HDL。其他例子将在本书其他地方提到。然而,这里的描述是不完整的,可能需要一些额外的引用。例 2-11 展示了如图 2-39 中有两个异或门和三个与非门的全加器的结构模型。Verilog 模型从关键字“module”开始,并且包含模块名(例如 full_adder),然后是输入和输出接口的列表(例如 a、b、cin、s 和 cout)。模型描述以关键字“endmodule”结束。接口列表可以以任何排序出现,但是必须声明为“input”或者

[79]

“output”。那些不声明为输入或者输出的信号名就被认为是本地的信号，当结构化设计时会当成线使用。例如，例子中的三个信号 out1、out2 和 out3 都是本地的。

例 2-11 如图 2-39 中的全加器的 Verilog 结构模块：

```
module full_adder
(
    input a, b, cin,
    output s, cout
); //defines a module's name and its interface signals

wire out1, out2, out3; //defines local signal names

xor    x1(out1, a, b);
xor    x2(s, out1, cin);
nand   n1(out2, out1, cin);
nand   n2(out3, a, b);
nand   n3(cout, out2, out3);

endmodule
```

标准逻辑门也称为原始逻辑门，它们可以被 Verilog 编译器所识别，不用重新描述。例子中的 x1、x2、n1、n2 和 n3 是可选的，并且是实例化的异或门和三个与非门给定的名称。每一个实例化原始逻辑门最左边的参数为输出，其余的都是输入。例如，信号 out1、s、out2、out3 和 cout 都是输出，所以它们在参数的最左边。原始逻辑门可以被一个或者多个输入参数所实例化，这依赖于它的类型。例如，一个三输入的原始与非门可能包含一个输出（最左边）和三个输入参数。

模块可以用任何顺序进行实例化，类似于我们在屏幕上使用图形设计工具进行实例化的过程（例如图 2-39）。模块的互连情况由其端口列表确定。例如，第一个实例化原始异或门以 out1 作为其输出接口，第二个原始异或门用 out1 作为其输入接口。这意味着有一条线连接两个 out1 接口。类似地，out2 和 out3 信号分别用一条线连接。这些信号都被声明为“线”。

如例 2-12 中的例子，测试台模块也用 HDL 进行描述，其功能是测试电路模块，例如例 2-11 中的全加器模块。测试台模块没有输入和输出接口。我们可能需要 `include 指令（取决于设计工具）来导入另一个模块中非原始但已经用 HDL 创建的模块——例如，测试台。通常每一个被导入的模块可以被实例化一次或多次，在调试前根据需要创建目标电路模块。在例 2-12 中，“full_adder” HDL 模块被导入且在调试中被实例化了一次。

80

初始块用于列出“full_adder”模块的测试容器。所有初始块中的语句都必须按照顺序处理。初始块或者永久块（稍后讨论）中所有在赋值运算符（例如 =）左边的变量都要声明为 reg 类型。当描述组合电路时，reg 类型没有具体的意义。然而 reg 类型在设计时序电路时将会非常重要。

例 2-12 测试例 2-11 中全加器结构模块的测试台：

```
`include "full_adder.v" //input FA's description from local folder
module tester();
reg a, b, cin;

wire s, cout;

full_adder fal(a, b, cin, s, cout); //instantiate FA

initial begin //start the test
$display("Time a b cin cout s"); //header for the output
$monitor ("%4d %b %b", $time, cout, s);
a = 0; b = 0; cin = 0; $display("%4d %b %b %b", $time, a, b, cin);
```

```
//test 1
#1 //simulate
a = 1; b = 1; cin = 0; $display("%4d %b %b %b", $time, a, b, cin);
//test 2
#1 //simulate
a = 1; b = 1; cin = 1; $display("%4d %b %b %b", $time, a, b, cin);
//test 3
#1 //simulate
$finish; //stops simulation
end
endmodule
```

当使用没有调试功能的设计工具时，\$display 语句可以用来输出一个或者多个输入信号的值。另一方面，\$monitor 语句在仿真运行期间只输入一次来跟踪一个或者多个信号（输入或者输出）。每一次 \$monitor 语句中列出信号的值发生改变，语句就会被执行来输出信号值。\$display 语句和 \$monitor 语句句法是一样的，与 C 编程语言中的 “printf” 语句相似。输出格式 “%d”、“%h”、“%o” 和 “%b” 可以分别用于表示十进制、十六进制、八进制和二进制的数值。附加输出格式，例如 “%s” 和 “%f” 用于表示字符串和浮点数值。

仿真的时间步是使用符号 “#” 后跟着代表仿真时间长度的整数表示。如果假定一个模块没有延迟，功能仿真将假定每个模块都没有传输延迟，并且在一个仿真时间（#1）内可以产生输出。

81

如果使用 Synopsys 设计工具，调试例 2-12 中 FA 模块的仿真输出如下所示。我们可以看到当 $a = 1, b = 1, cin = 1$ 时，\$monitor 语句在仿真时间 = 2 时输出 $s = 1, cout = 1$ 。

```
Chronologic VCS simulator
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12;

Time  a    b    cin cout  s
0     0    0    0     0    0
0
1     1    1    0     1    0
1
2     1    1    1     1    1
2

$finish called from file "tester2.v", line 18.
$finish at simulation time 3
```

VCS Simulation Report

```
Time: 3
CPU Time: 0.460 seconds;    Data structure size: 0.0Mb
```

“=” 符号被称为块赋值。所有的包含在原始块或者永久块中的块赋值语句按一定的顺序进行评估，这个过程类似于编程语言。另一方面，非块赋值（稍后讨论）用符号 “<= ” 表示，并且与其他在原始块或者永久块中的非块赋值语句同时执行。包含在原始块中的 HDL 语句只被评估一次，就像真正的电路一接通电源就开始工作；而包含在永久块中的 HDL 语句只要电路是在仿真阶段中就会被评估。

Verilog 还可以支持 “for 循环”、“case”（switch）、“forever” 和其他控制语句。然而，不是所有的 Verilog 语句都可以进行合成。如例 2-13 所示的测试台，用 for 循环完全测试了 FA 模块。在例子中，信号 a、b 和 cin 都被声明为 1 位 reg，而用在 for 循环中的变量 k，用小尾数顺序声明为 4 位 reg。语句 “reg [0:3] k;” 将 k 定义为大尾数顺序。多位变量可以作为一个组或者按位进行操作引用。例如，在例子中 k[2] 表示变量 k 的第三位，

k[0] 表示 k 中的最低有效位 (LSB)。

例 2-13 完全测试例 2-11 中 FA 结构模块的测试台模块：

```
'include "full_adder.v"
module tester();
reg a, b, cin;

reg [3:0] k;
wire s, cout;
full_adder fa1(a, b, cin, s, cout);

initial begin
$display("Time a b cin cout s");
$monitor ("%4d %b %b", $time, cout, s);

for (k = 0; k <= 7; k = k+1) begin
    #1 a = k[2]; b = k[1]; cin = k[0]; $display("%4d %b %b %b",
$time, a, b, cin);
end
#10 $finish; //stops simulation
end
endmodule
```

82

在例 2-13 的测试台中，在每一个仿真步骤，都有新的数值赋给输入信号 *a*、*b* 和 *cin*。输入信号也可以在每一个仿真步骤中显示。在每一个仿真步骤结束后，\$monitor 语句自动地显示输出信号的数值，展示一个或者多个信号值的变化。仿真运行中的输出将会在下一步展示。当测试容器为 *a* = 0、*b* = 0 和 *cin* = 1，*a* = 0、*b* = 1 和 *cin* = 0，*a* = 1、*b* = 0 和 *cin* = 1，*a* = 1、*b* = 1 和 *cin* = 0 时，信号 *s* 和 *cout* 不改变，\$monitor 语句不会显示输出信号。

Chronologic VCS simulator
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12;

| Time | a | b | cin | cout | s |
|------|---|---|-----|------|---|
| 0 | | | | x | x |
| 1 | 0 | 0 | 0 | | |
| 1 | | | | 0 | 0 |
| 2 | 0 | 0 | 1 | | |
| 2 | | | | 0 | 1 |
| 3 | 0 | 1 | 0 | | |
| 4 | 0 | 1 | 1 | | |
| 4 | | | | 1 | 0 |
| 5 | 1 | 0 | 0 | | |
| 5 | | | | 0 | 1 |
| 6 | 1 | 0 | 1 | | |
| 6 | | | | 1 | 0 |
| 7 | 1 | 1 | 0 | | |
| 8 | 1 | 1 | 1 | | |
| 8 | | | | 1 | 1 |

\$finish called from file "tester.v", line 16.
\$finish at simulation time 18

VCS 仿真报告

Time: 18
CPU Time: 0.390 seconds; Data structure size: 0.0Mb

83

2.10.2 传输延迟仿真

在每一个初始逻辑门在实例化的过程中也可以包含可选的传输延迟。这就提供了更现实

的功能仿真。例 2-14 描述了原始与非门有 1ns 延迟，原始异或门有 3ns 延迟的全加器。由此可得： $\Delta s = 6ns$ ， $\Delta cout = 5ns$ 。

例 2-14 有延迟的用原始逻辑门实现的 FA 的结构化模块：

```
`timescale 1ns/100ps
module full_adder
(
    input a, b, cin,
    output s, cout
);

wire out1, out2, out3;

xor    #3 x1(out1, a, b);
xor    #3 x2(s, out1, cin);
nand   #1 n1(out2, out1, cin);
nand   #1 n2(out3, a, b);
nand   #1 n3(cout, out2, out3);

endmodule
```

编译器语句 `timescale 表示仿真中应用的时间范围。例 2-14 中的 `timescale 语句定义了有 100ps（皮秒）增量的 1ns。然而，这个时间范围仅仅是一个估计，仿真结果并没有提供真正的时间数据。

例 2-15 中的测试台包含了分别在仿真时间 0 和 10 进入的两个测试容器。如下仿真输出所示，最初信号 *cout* 和 *s* 的值都为未知（“x”）。对于测试容器在仿真时间 = 0 时的 *a* = 0、*b* = 0 和 *cin* = 1，语句 \$monitor 有输出，在仿真时间 = 5 时，输出 *cout* = 0，在仿真时间 = 6 时，输出 *s* = 1，与预期相符。对于测试容器在仿真时间 = 10 时的 *a* = 0、*b* = 1 和 *cin* = 1，在仿真时间 = 15 输出 *cout* = 1，在仿真时间 = 16，输出 *s* = 0，与预期相符。这两个测试容器暴露出了电路最严重的延迟情况。

例 2-15 有延迟的 FA 全加器模块的测试台：

```
`include "full_adder.v"
`timescale 1ns/100ps
module tester();
reg a, b, cin;

wire s, cout;

full_adder fa1(a, b, cin, s, cout);
initial begin
    $display("Time a b cin cout s");
    $monitor ("%4d %b %b", $time, cout, s);
    a = 0; b = 0; ci = 1; $display("%4d %b %b %b", $time, a, b, ci);
    #10
    a = 0; b = 1; ci = 1; $display("%4d %b %b %b", $time, a, b, ci);
    #10 $finish;
end
endmodule
```

Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.

| Time | a | b | c | cout | s |
|------|---|---|---|------|---|
| 0 | 0 | 0 | 1 | | |
| 0 | | | | x | x |
| 5 | | | | 0 | x |
| 6 | | | | 0 | 1 |
| 10 | 0 | 1 | 1 | | |


```
15          1    1
16          1    0
$finish at simulation time 200
```

VCS 仿真报告

```
Time: 20000 ps
CPU Time: 0.040 seconds; Data structure size: 0.0Mb
```

非原始模块不能在有延迟信息的情况下进行仿真。对于这些模块，延迟信息是由模块内部的延迟数值决定的。然而，我们可以在实例化过程中用参数化的延迟去重写延迟信息。

2.10.3 行为建模

Verilog 中最基础的行为描述是赋值语句。它使用符号“~”、“&”、“|”和“^”直接进入一个布尔表达式分别去按位表达非、与、或和异或操作。与非、或非和异或非操作也可以分别用符号组合“&~”、“|~”和“^~”或者“^~”表示。表 2-13 总结了 Verilog HDL 中的操作符。例 2-16 使用赋值语句展示了 1 位 2-1 MUX。

表 2-13 Verilog HDL 操作符总结

| 优先级 | 操作符类型 | 符 号 | 示 例 |
|-----|-------|----------------|--|
| 最高 | 一元 | +, -, !, ~ | + a, - a (a 取反), !a (逻辑非), ~a (按位取反) |
| | 指数 | ** | a ** 3 (a 的立方) |
| | 1 类算术 | *, /, % | a * b (乘), a / b (除), a % b (取余) |
| | 2 类算术 | +, - | a + b (加), a - b (减) |
| | 位移: | | |
| | 逻辑的 | <<, >> | a << 2 (左移 2 位) |
| | 算术的 | <<<, >>> | a >>> 3 (包括符号位右移 3 位) |
| | 关系的 | <, <=, >, >= | a >= b (a 大于等于 b) |
| | 等值: | | |
| | 逻辑的 | ==, != | a == b 如果 a 与 b 相同, 不包含 x 和 z |
| | 判断 | ===, !== | a === b 如果 a 与 b 相同, 包含 x 和 z |
| | 按位: | | |
| | 基础的 | &, , ~, ^ | a & b (与), a b (或), ~a (非), a ^ b (异或) |
| | 组合的 | &~, ~, ~^, ^~ | a &~ b (与非), a ~ b (或非), a ~^ b (异或非), a ^~ b (异或非) |
| | 逻辑 | &&, , ! | a && b (与), a b (或), !a (非) |
| | 最低 | 组合 | ?: (a >= b) ? a - b : b - a |

例 2-16 用赋值语句描述 1 位 2-1 MUX 的行为模块：

```
module mux1bit
(
    input s,
    input x, y,
    output r
);
assign r = ~s & x | s & y;
endmodule
```

描述电路行为最常使用的语句叫永久块。我们用关键字“always”后面跟着符号 @

和一个称为块的**敏感列表**的信号名列表来声明永久块，和初始(initial)块类似，永久块也包含 begin-end 块。例 2-17 展示了使用“if-else”语句表示 1 位 2-1 MUX 的另一种行为模型。因为 r 信号依赖于信号 s 、 x 和 y ，这些信号都会包括在永久块的敏感列表中。

例 2-17 使用“if-else”语句表示 1 位 2-1 MUX 的行为模型：

```
module mux1bits
(
    input s,
    input x, y,
    output reg r
);
    always@(s or x or y)
    begin
        if (s == 1'b0)
            r = x;
        else
            r = y;
    end
endmodule
```

86

与初始块类似，永久块中所有在赋值符号(=)左边的变量必须被声明为 reg 类型。声明一个同时为 output 和 reg 类型的输出变量的正确方法为用语法 output reg 来声明，如例 2-17 所示。

语法 1'b0 用来表示 1 位二进制数。其他表示数字的例子有 5'b11111、8'hFF 和 9'o777，分别表示 5 位二进制数 $(11111)_2$ 、8 位十六进制数 $0xFF$ 、9 位八进制数 $(777)_8$ 。这里“8”用来表示一个八进制数。

依赖于编译器的版本，也有另外的语法来描述敏感列表。例如，“always@(s or x or y)”、“always@(s, x, y)”、“always@(*)”或者“always@*”都是可接受的表示组合电路敏感列表的语法。此外，在组合永久块中用 * 表示敏感列表是最常见的语法，可以允许编译器决定敏感变量的列表。敏感列表中遗漏变量可以导致组合电路行为的错误。

例 2-18 用 case 语句表达其真值表展示了 FA 全加器模块的行为。在例子中，大括号({})表示一个级联。“default”情况也是考虑遗漏情况的正常操作。可接受的信号值为 0、1、 x (未知) 和 z (高阻抗, Z)。作为一种选择，“casex”忽略 x 和 z 信号值并且把它们视为无关项。

例 2-18 用“case”语句表示的 FA 全加器行为模块：

```
module full_adder
(
    input a, b, cin,
    output reg s, cout
);
    always@(a or b or cin)
    begin
        case ({a, b, cin})
            3'b000: begin s = 0; cout = 0; end
            3'b001: begin s = 1; cout = 0; end
            3'b010: begin s = 1; cout = 0; end
            3'b011: begin s = 0; cout = 1; end
            3'b100: begin s = 1; cout = 0; end
            3'b101: begin s = 0; cout = 1; end
            3'b110: begin s = 0; cout = 1; end
```

87

```
3'b111: begin s = 1; cout = 1; end
default: begin s = 0; cout = 0; end
endcase
end
endmodule
```

例 2-19 展示了有低电平使能信号 4 位三态缓存行为描述，这里的 4'bz 表示一个 4 位高阻抗 Z 值。

例 2-19 4 位三态缓存的行为模块：

```
module tristate4bits
(
    input _e, //declared active-low
    input [3:0] x,
    output reg [3:0] r
);
always@(*)
begin
    if (_e == 1'b0)
        r = x;
    else
        r = 4'bz; //or r = 4'bzzzz;
end
endmodule
```

2.10.4 综合与仿真

例 2-18 中 FA 全加器 HDL 模型运用了 Altera Quartus II 设计和综合工具来进行综合 [4]。在综合过程中，使用了一种叫 EP4CGX15BF14A7 1.2V 的 Altera Cyclone IV GX 系列中的可编程芯片。综合后的电路可以用 Altera ModelSim 仿真工具来进行仿真。图 2-40 展示了有两个 CLB 来实现和值和进位输出的表达式的综合电路。在这个例子中，输入 a、b 和 cin，以及输出 s 和 cout 都使用缓存来防止可能的扇出问题。图 2-41 展示了有 8 个测试容器波形的综合电路的仿真。

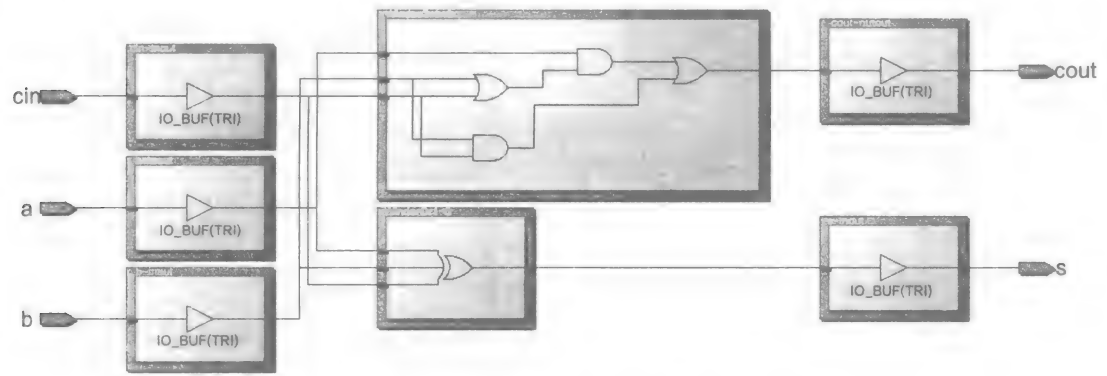


图 2-40 例 2-18 中 FA 全加器行为模块的综合电路

类似的，例 2-19 中的 4 位三态缓存模型也可以用 Altera 设计、综合和仿真工具进行综合和仿真。综合电路如图 2-42 所示。因为使能信号 _e 是低电平信号，所以不需要附加的非门来操作低电平使能的三态缓存。有两个测试容器的综合电路的仿真图如图 2-43 所示。

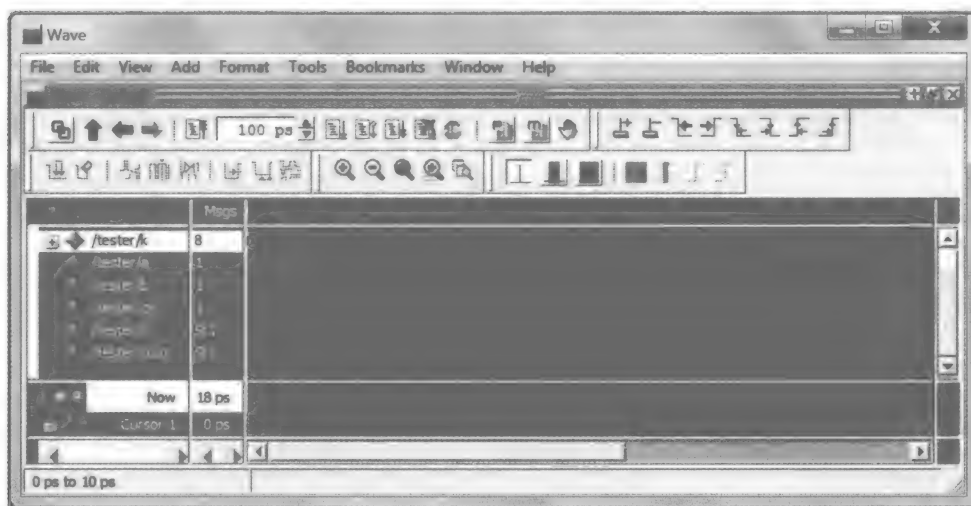


图 2-41 图 2-40 中综合 FA 全加器电路的门级仿真

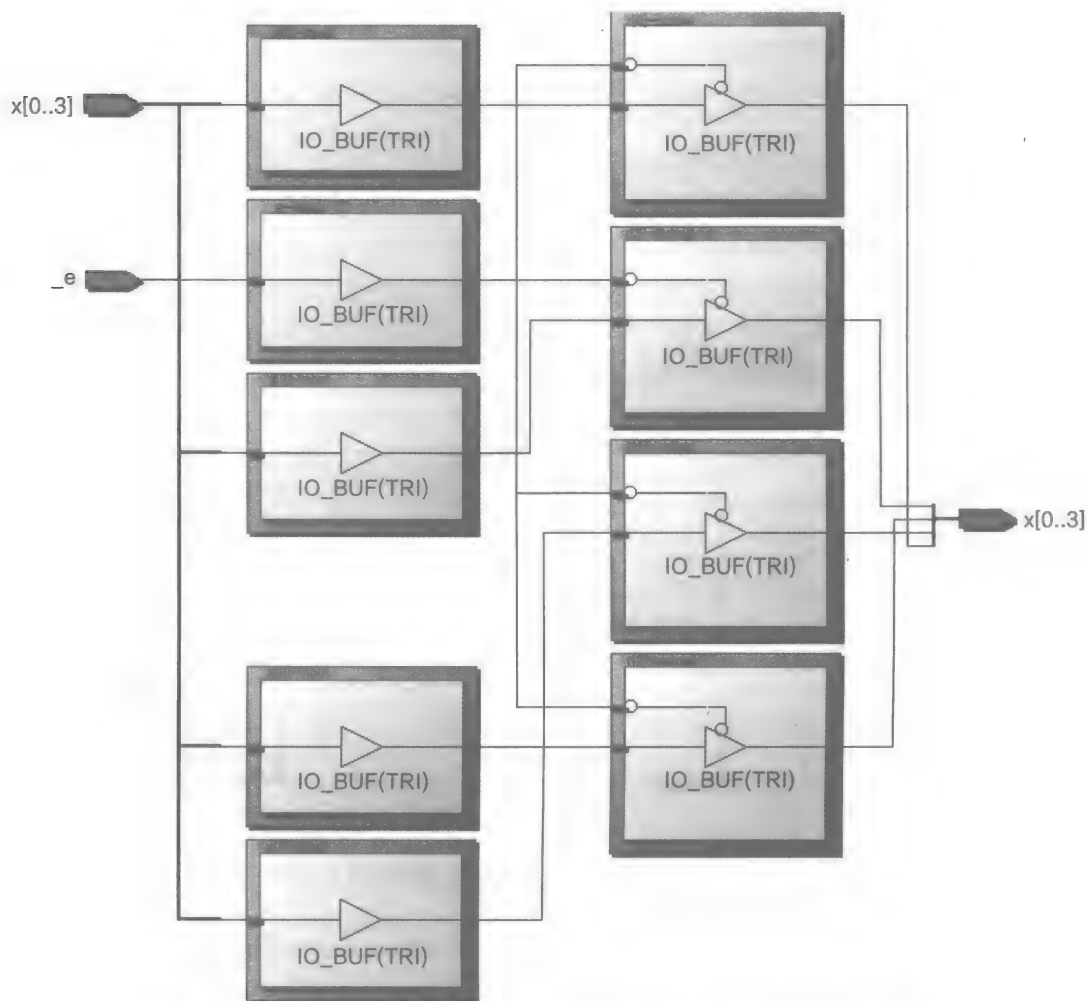


图 2-42 用 Altera Quartus II 设计工具合成的 4 位三态缓存综合电路

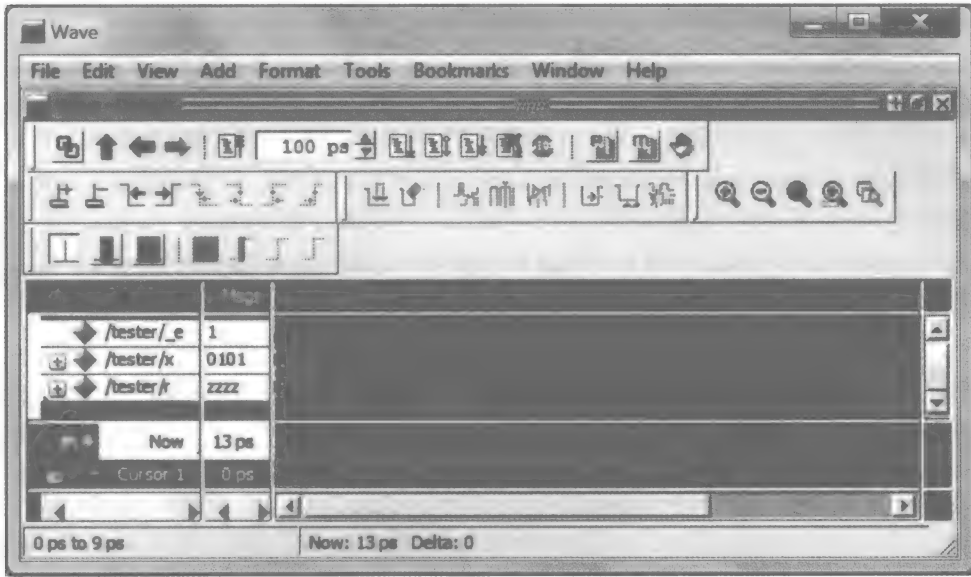


图 2-43 图 2-42 中 4 位三态缓存的仿真输出

参考文献

1. Espresso, <http://diamond.gem.valpo.edu/~dhart/ece110/espresso/tutorial.html>.
2. SN54/74LS245 Octal Bus Transmitter/Receiver from 7400 chip series.
3. Xilinx FPGAs, <http://www.xilinx.com/>.
4. Altera, <http://www.altera.com>.
5. EasyFPGA, <http://www.easyfpga.com/>.
6. Duncan Buel, Tarek El-Ghazawi, Kris Gaj, and Voldymyr Kindratenko, "High-performance reconfigurable computing," *IEEE Computer*, March 2007, pp. 23–27.
7. LogicWorks, Digital design schematic tool, Pearson Publishing, <http://www.pearsonhighered.com/>.

练习

- 2.1 当 $x = 1, y = 0, z = 1$ 和 $x = 1, y = 1, z = 0$ 时, 计算 $f = \overline{xy} + yz$ 。
- 2.2 当 $_c = 0, x = 1$ 和 $_c = 1, x = 1$ 且 $_c$ 是一个低电平信号时, 计算 $y = _c \bar{x} + _c x$ 。
- 2.3 分别求出 $f = \overline{xy}$ 和 $g = \bar{x} + \bar{y}$ 的真值表, 并证明德摩根定律 $\overline{xy} = \bar{x} + \bar{y}$ 。它们的真值表一样吗?
- 2.4 分别求出 $f = \overline{x + y}$ 和 $g = \bar{x} \bar{y}$ 的真值表, 并证明德摩根定律 $\overline{x + y} = \bar{x} \bar{y}$ 。它们的真值表一样吗?
- 2.5 画出 $f = \overline{xy} + yz$ 的电路图, 然后按照本书中介绍的步骤将其转换成用与非门表示的电路。
- 2.6 当 $x = 1, y = 0, z = 1$ 和 $x = 1, y = 1$ 和 $z = 0$ 时, 计算 $f = (x + y)(\bar{y} + z)$ 。
- 2.7 画出 $f = (x + y)(\bar{y} + z)$ 的电路图, 然后按照本书中介绍的步骤将其转换成用或非门表示的电路。
- 2.8 给出表达式 $f = \overline{xy} + yz$ (SOP 表达式), 求出与其等价的 POS 表达式。提示: 首先运用规则“ f 的 POS 表达式 = \bar{f} 的 SOP 表达式的取反”找出 \bar{f} 的 SOP 表达式。
- 2.9 对的 SOP 表达式 $f = \overline{xy} + yz$ 运用对偶定律求出 f 的 POS 表达式。
- 2.10 假设我们需要建立函数 $Y = 2X + 3$, 其中在硬件中 X 代表 3 位无符号数值 $(x_2x_1x_0)_2$, Y 代表 5 位数值 $y_4y_3y_2y_1y_0$ 。当输入位为 x_2, x_1 和 x_0 , 输出位为 y_4 到 y_0 时, 建立真值表。然后对于 y_2 (也可以对其他输出重复这些工作):
 - a. 求出输出位 y_2 的规范 SOP 表达式。

- b. 写出 y_2 的最小项。
- c. 用 K 图找出 y_2 的最小 SOP 表达式。
- d. 画出 y_2 的与非门最小电路。
- e. 比较规范 SOP 表达式和最小 SOP 表达式所需要的三极管数目。

2.11 用 y_2 的 POS 表达式重复 2.10 中的步骤。

2.12 重复 2.10 中 $b \sim d$ 步, 但是这次对 X 使用 3 位 2 的补码值且输出 y_4 。

2.13 重复 2.12 中 $b \sim d$ 步, 但是这次使用 y_4 的 POS 表达式。

2.14 用 K 图的方法找出下列函数的最小 SOP 表达式:

a. $f(w, x, y, z) = \Sigma(0, 2, 8, 10) + \Sigma_d(12, 14)$

b. $g(a, b, c, d) = \Sigma(5, 7, 13, 15) + \Sigma_d(6, 14)$

c. $h(w, x, y, z) = \Pi(0, 2, 8, 10) + \Pi_d(12, 14)$

d. $t(a, b, c, d) = \Pi(5, 7, 13, 15) + \Pi_d(6, 14)$

2.15 找出 2.14 中函数的最小 POS 表达式。

2.16 用 Espresso 软件生成函数 $Y = 2X + 3$ 所有输出位的最小 SOP 表达式, 这里 X 是小于 10 的 4 位无符号数值。当 X 数值为 10 ~ 15 时, 当作无关项忽略掉。

91 2.17 重复练习 2.16, 但是用 4 位 2 的补码数值, 且 $-5 \leq X \leq 5$, 且忽略掉 $X \leq -5$ 和 $X \geq 5$ 。

2.18 给定函数 $Y = X \bmod 7$, $X = x_3x_2x_1x_0$, 且为 4 位无符号输入, $Y = y_2y_1y_0$, 且为 3 位无符号结果, 建立 Y 的真值表, 并且决定 y_2 、 y_1 和 y_0 的 SOP 和 POS 表达式。

2.19 用逻辑化简算法求出函数 $Y = X - 3$ 中的 y_0 的最小 SOP 表达式, 这里 $X = x_3..x_0$, $Y = y_3..y_0$, 它们都是 4 位 2 的补码数值。

2.20 用逻辑化简算法求出 $y = \Sigma(2, 3, 6, 9, 10, 13)$ 的最小 SOP 表达式。

2.21 $f(a, b, c, d) = \Sigma(1, 3, 5, 7, 10, 11, 14, 15)$ 的素蕴含为 $\bar{a}d + ac$ 和 cd 。最小表达式 $f = \bar{a}d + ac$ 的时序图如图 2-25 所示。画出非最小表达式 $f = \bar{a}d + ac + cd$ 的电路图, 要包含其所有素蕴含, 且标出所有中间信号。当其输入从 $acd = 111$ 到 $acd = 011$ 时, 画出电路的时序图。所画的电路会产生故障吗?

2.22 函数 $f(a, b, c, d) = \Pi(0, 2, 4, 6, 8, 9, 12, 13)$ 的最小 POS 表达式有两个基本的素蕴含 $(a + b)$ 和 $(\bar{a} + c)$, 以及一个非基本的素蕴含 $(c + d)$ 。

a. 当输入从 $acd = 000$ 改变为 $acd = 100$ 时, 画出最小表达式 $f = (a + d)(\bar{a} + c)$ 的时序图。所画电路会产生故障吗?

b. 当输入从 $acd = 000$ 改变为 $acd = 100$ 时, 画出非最小表达式 $f = (a + d)(\bar{a} + c)(c + d)$ 的时序图。考虑到 f 包括其所有的素蕴含。这里会有 1- 冒险吗?

2.23 设计一个二输入的线-或门(提示: 运用德摩根定律)。

2.24 求出图 2-31 中 2-1 MUX 的 POS 表达式。

2.25 用 4-1 MUX 设计函数 $f(w, x) = \Sigma(0, 2)$ 。

2.26 用 2-4 译码器连接 4 个模块, 每个模块输出一位到一位总线上。每一次只有一个模块可以将数据放置在总线上。有时任意模块将不允许在总线上放置数据。给出细节。

2.27 用与非门设计图 2-35 中的 3-2 译码器电路。

2.28 假设在问题 22 中的译码器可以循环地在每 10ns 中激活每个输出信号, 并在每 10ns 中允许每个模块输出一位。问在每个模块中以字节传输数据的最高速度是多少? 总线的最大带宽是多少?

提示: 传输速度和带宽单位为字节/秒。传输数据的最高速度是每秒钟模块可以传送数据的最大

字节数（单位为 KB、MB 等）。最大总线带宽是总线在每秒钟可以传输的字节最大值。

2.29 对于函数 $f = x\bar{y} + yz$ ，建立变量 x 、 y 和 z 的 Verilog 模型并进行仿真，分别使用：

- a. 非门、与门和或门结构描述。
- b. 非门和与非门结构描述。
- c. 有 1ns 延迟的非门、与非门和有 2ns 延迟的与门和或门结构描述。
- d. 用 “assign” 语句进行行为描述。
- e. 用 “always” 语句进行行为描述。

92

2.30 对于函数 $f = (x + y)(\bar{y} + z)$ ，建立变量 x 、 y 和 z 的 Verilog 模型并进行仿真，分别使用：

- a. 非门、与门和或门结构描述。
- b. 非门和或非门结构描述。
- c. 有 1ns 延迟的非门、或非门和有 2ns 延迟的与门和或门结构描述。
- d. 用 “assign” 语句进行行为描述。
- e. 用 “always” 语句进行行为描述。

2.31 建立 1-4 MUX 的 Verilog 行为描述并进行仿真，符合以下要求：

- a. 用 “if-else” 语句。
- b. 用 “case” 语句。

2.32 建立 2-4 译码器的 Verilog 行为描述并进行仿真。使用正确的极性来标记信号名（例如， $_x$ 可以用于表示低电平信号， x 可以用于表示高电平信号）。

- a. 高电平输入和高电平输出，使用 “always” 语句。
- b. 高电平输入和低电平输出，使用 “always” 语句。

2.33 建立 3-2 译码器的 Verilog 行为描述并进行仿真。使用正确的极性来标记信号名（例如， $_x$ 可以用于表示低电平信号， x 可以用于表示高电平信号）。

- a. 高电平输入和高电平输出，使用 “always” 语句。
- b. 低电平输入和高电平输出，使用 “always” 语句。

93

组合电路：大型设计

3.1 简介

在前一章中介绍的设计技术只适用于很少输入数目的组合电路。有很多输入的组合电路必须以不同的方式设计。例如，考虑有 $n = 32$ 个输入的组合电路。其真值表可能会有超过 40 亿行——对于第 2 章介绍的设计电路方法来说，这个数目特别巨大。再有，大型电路肯定会遇到设计扇入和扇出的需求。这就要求一种自顶向下的方法，将大型组合电路设计问题反复地分成较小的问题，直至小到可以用第 2 章所学的设计技术来解决为止。然后大型电路通过将较小型电路模块进行组装来设计完成。

具有加法、减法、乘法和除法等初级算术操作的电路是大型电路应用于处理器的例子。类似于软件解决方案可以实现需要不同处理器时间和存储器使用情况的代替算法，大型组合电路可以用不同总量的硬件实现（例如三极管数量）。具有越多的三极管数量的电路解决方案，通常意味着具有更小的电路延迟来执行更高数量的逻辑操作，但也更耗能量。

通常，更多的硬件意味着更少的处理时间。包含快速算法模块的 CPU 有望更快地运行；多核处理器有望比单核处理器运行任务速度更快，等等。

在这章中，我们提供了算术电路的设计实例。特别地，我们将讨论通常称为快速加法器的设计，也展示减法器、2 的补码加法器、算术逻辑单元（ALU）、乘法器和除法器的设计。这章同样也会展示 IEEE 浮点（FP）数标准和算术。

自顶向下的设计方法

自顶向下的方法，也叫分层的方法，是指一种设计流程，如包含父结点和叶结点的树结构。在根结点的大型设计问题可以依次在叶结点划分为较小的设计问题。根结点的问题第一次将被分为较小设计问题为子结点。如果需要，在每一个子结点（现在为父结点）上的较小设计问题又可以分为更小的设计问题。这个处理过程一直持续直到每一个在叶结点的设计问题足够小并且拥有更少的输入。

对于叶结点上的每一个设计问题，第 2 章中的技术都可以用来设计电路。这些电路随后将连续组合成目标大型组合电路。设计可能需要一个或多个较小电路的副本。最终电路必须无任何扇入和扇出问题。

在设计流程中的每一步，有很多种方法和算法都会进行检查，对如电路延迟和逻辑门数量进行权衡分析，然后选出最佳的解决方案来满足整个设计要求。通常，一位并行和一位串行设计分区技术（稍后定义）从根结点开始被用于每一个父结点。

当设计问题被分成较小的设计问题时，分区被称为位并行的。例如，生成 n 位与或者 n 位和的组合 AND/ADD 电路模块的设计可以视为三个分开的设计问题： n 位的位与逻辑、 n 位的加法器和 n 位 2-1 的多路选择器（MUX），当 $n = 8$ 时如图 3-1 所示。MUX 选择位与逻辑的结果或者加法器的和值来作为输出。虽然位与逻辑、加法器和 MUX 都只是实现单一的

功能，但是对于第 2 章所学的方法来说，还是很大型的设计问题。这三个模块都必须再分为更小的设计问题。

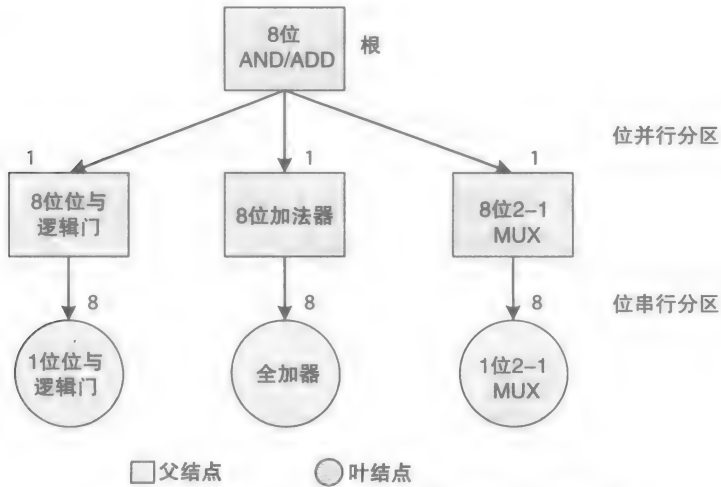


图 3-1 自顶向下位并行和位串行设计分区实例

96

位串行方法用于将设计问题分成有更少输入的较小问题。例如， n 位输入的大型电路设计问题可以分为称为片的 k 位输入设计问题，这里 k (最好) 能整除 n 。每片都可以执行一个或者多个函数操作，但只能对较少的输入位进行操作。例如，一个 8 位加法器可以用 8 个全加器 (FA) 进行设计；8 位 2-1 MUX 可以用 8 个 1 位 2-1 MUX 进行设计。在图中，每个结点旁边的数字表示了片的数量。在 k 位片不会违反所用逻辑门的扇入扇出限制时，常数 k 可以作为一个选择。

在用位串行方法设计 8 位 AND/ADD 模块时，我们可以使用 8 片 1 位 AND/ADD 片或者 4 片 2 位 AND/ADD 片来进行设计，如图 3-2 所示。通常 n 位模块可以用 n 个 1 位、 $n/2$ 个 2 位、 $n/4$ 个 4 位等来设计。每片需要输出附加信号——例如，当选择加法函数时，连接片所需要的进位输出位。为了保证最小传输延迟，位串行可以被用真值表和其他设计方式化简成 SOP 或者 POS 表达式进行模块化。

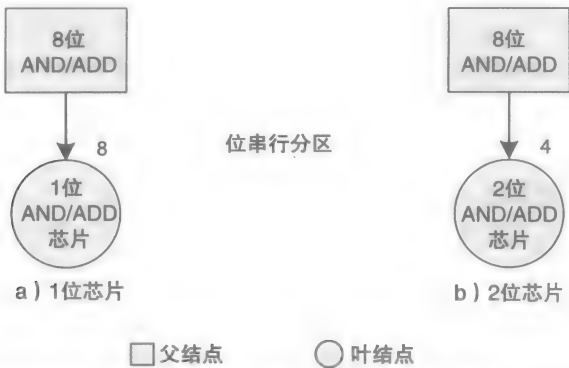


图 3-2 自顶向下，位串行设计分区实例：a) 8 个 1 位芯片；b) 4 个 2 位芯片

使用位并行还是位串行方法，在每一步的使用中都会对结果电路的最大传输延迟和所需

逻辑门数目造成影响。这一章展示了一些常见的大型组合电路。

3.2 算术函数

加法、减法、乘法和除法是 4 种基本算术函数。更多复杂的函数，例如求平方根、指数和正弦值等，这些对浮点 (FP) 数操作的函数都会用到基本的 4 个函数来产生输出。随着这些年集成芯片 (IC) 中三极管的数量不断增加，越来越多的算术函数可以在硬件上实现。例如，早期的微处理器只能在硬件上实现加法和减法功能，其他的功能只能在软件上实现。现在的许多计算器还是有一小部分功能在硬件上实现而大部分在软件上实现。现代微处理器通常包括叫作整数单元 (IU) 的整数算术电路和浮点单元 (FPU)。通常，一个现代微处理器包括多个 IU 和 FPU，也包括在第 1 章中讨论的整数和浮点数单指令多数据流 (SIMD) 单元。

3.3 加法器

n 位整数加法器输入两个 n 位数值和一个可选的输入进位 (c_{in}) 来产生一个 n 位的和及最后进位输出 (c_{out})。如图 3-3 所示，从右 (例如最低有效位) 开始，两位和先前的进位相加产生一个和及一个进位 (0 或 1) 来进行下两位相加。在图中，初始的 c_{in} 假设为 0。算法重复直到最后两位相加并产生最后的和及最后的进位输出 c_{out} 停止。

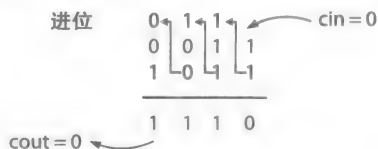


图 3-3 4 位二进制加法图

在图中，最先的两位 (都为 1) 和 $c_{in} = 0$ 相加得到 0 且进位为 1。下一个两位，也都为 1，且先进位为 1，相加得到和及进位都为 1。其他位也进行类似的加法操作。这个简单的加法算法可以看成是多个 1 位加法步骤，每一个都是一个全加器 FA，如图 3-4 中第 i 步所示。在算法的每个步骤中，两个 1 位输入 a_i 和 b_i 和先前进位 c_{i-1} 相加产生和 s_i 和下一个进位 c_i 。对于 n 位数，这个过程需要重复 n 次及 $i = 0, 1, \dots, n-1$, $c_{-1} = c_{in}$, $c_{out} = c_{n-1}$ 。



图 3-4 在 FA 中查看每一步二进制加法

3.3.1 进位传输加法器

在图 3-4 中实现简单加法算法的 n 位加法器使用 n 个 FA 片串联而成的，如图 3-5 所示。从最低有效位进位开始，进位 c_0 到 c_{n-1} 在每一步中产生一个，类似于手工加法运算。这种加法器又叫进位传输加法器 (CPA)，因为进位是从一个 FA 片到下一个 FA 片一个一个产生的。当进位在电路中波纹传输，就像在进位链中进位反馈给下一个 FA 片时，这种加法器叫行波进位加法器 (RCA)。

进位传输加法器是加法器中最简单的电路，但有正比于进位数量的最长传输延迟。每一个进位信号都依赖于先前的进位信号； c_1 依赖于 c_0 ， c_2 依赖于 c_1 等 (原书有误——译者注)。公式 (3-1) 用于估计 n 位 CPA 的传输延迟， $\Delta CPA(n)$ 表示 n 位 CPA 的传输延迟， ΔFAc 和 ΔFAs 分别表示 FA 中进位及和的传输延迟。

$$\Delta CPA(n) = (n-1) * \Delta FAc + \Delta FAs \quad (3-1)$$

公式 (3-2) 表示在与非门延迟 $\Delta NAND = 0.1ns$ 和如第 2 章公式 (2-8) SOP 表达式中，

FA 的进位及和延迟分别为 $\Delta FAc = 0.2\text{ns}$ 及 $\Delta FAs = 0.3\text{ns}$ 的 n 位 CPA 传输延迟的计算过程。当 $n = 8$ 时, $\Delta CPA(8) = 1.7\text{ns}$ 。

$$\Delta CPA(n) = (n-1)(0.2\text{ns}) + 0.3\text{ns} = (0.2n + 0.1)\text{ns} \quad (3-2)$$

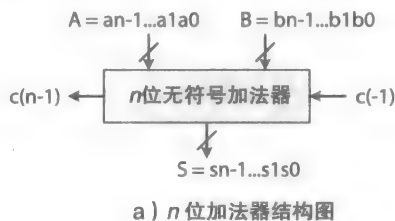


图 3-5 n 位加法器: a) 框图; b) n 位 CPA

99

3.3.2 先行进位加法器

当 n 很大时, CPA 的传输延迟可以变得过大。电路会产生一个非常长的进位产生链。然而, 我们也可以用更多的独立逻辑门来在更少的时间内并行产生进位。为了说明这点, 我们将定义两个变量传输 (p) 和产生 (g), 如下所示, 其中 i 为数位位置, $i = 0, 1, 2, \dots, n-1$ 。

$$\begin{aligned} p_i &= a_i \oplus b_i \\ g_i &= a_i b_i \end{aligned} \quad (3-3)$$

注意到所有的 p 和 g 位都分别可以并行地使用 n 个异或门和 n 个与门同时生成。由公式 (2-9)(第 2 章) 得:

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_{i-1} \\ c_i &= (a_i \oplus b_i) c_{i-1} + a_i b_i \end{aligned} \quad (3-4)$$

将公式 (3-3) 代入公式 (3-4) 得:

$$\begin{aligned} s_i &= p_i \oplus c_{i-1} \\ c_i &= g_i + p_i c_{i-1} \end{aligned} \quad (3-5)$$

公式 (3-6) 列出了前三个带有 p 、 g 变量及前一个进位的进位表达式。表达式清楚展示了进位的递归依赖性。

$$\begin{aligned} c_0 &= g_0 + p_0 c_{-1} \quad (\text{依赖于进位 } c_{-1}) \\ c_1 &= g_1 + p_1 c_0 \quad (\text{依赖于进位 } c_0) \\ c_2 &= g_2 + p_2 c_1 \quad (\text{依赖于进位 } c_1) \end{aligned} \quad (3-6)$$

如果每一个依次的进位可以代入后来的进位表达式中, 那么就可以并行地产生 c_1 、 c_2 等进位。进位 c_1 和 c_2 的产生过程展示如下:

$$\begin{aligned}
 c_0 &= g_0 + p_0 c_{-1} && \text{依赖于进位 } c_{-1} \\
 c_1 &= g_1 + p_1 c_0 \\
 &= g_1 + p_1 (g_0 + p_0 c_{-1}) && \text{代入 } c_0 \text{ 表达式} \\
 &= g_1 + p_1 g_0 + p_1 p_0 c_{-1} && \text{现在 } c_1 \text{ 依赖于进位 } c_{-1} \\
 c_2 &= g_2 + p_2 c_1 \\
 &= g_2 + p_2 (g_1 + p_1 g_0 + p_1 p_0 c_{-1}) && \text{代入 } c_1 \text{ 表达式} \\
 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{-1} && \text{现在 } c_2 \text{ 也依赖于进位 } c_{-1}
 \end{aligned} \tag{3-7}$$

c_1 和 c_2 最后的表达式都依赖于 c_{-1} ，这样它们就可以并行地产生了。然而相比于公式 (3-6)，每一个扩展表达式都需要更多的硬件去实现，包括逻辑门、更大的扇入和扇出需求。

100

p 和 g 位信号是用于定义快速在大量输入中计算进位的模式。图 3-6 中的例子展示了用 3 位输入 $A = a_2 a_1 a_0$ 和 $B = b_2 b_1 b_0$ 的计算过程。在实例 a 中， p_2 、 p_1 和 p_0 都为 1；这样，进位输入为 1 ($c_{-1} = 1$) 将传输输出为 $c_2 = 1$ ，因为在公式 (3-7) 中 $p_2 p_1 p_0 c_{-1} = 1$ 。在实例 b 中， $p_2 = 1$ ， $p_1 = 1$ 且 $g_0 = 1$ ；这样，进位输入为 0 (例如 $a_0 = 1$ 、 $b_0 = 1$ ，这样 $g_0 = 1$) 则产生输出 $c_2 = 1$ ，因为 $p_2 p_1 p_0 = 1$ 。其他的实例展示了其他情况，使得 c_2 为 1。在实例 c 中， $p_2 g_1 = 1$ ，所以 $c_2 = 1$ ，在实例 d 中， $g_2 = 1$ ，所以 $c_2 = 1$ 。

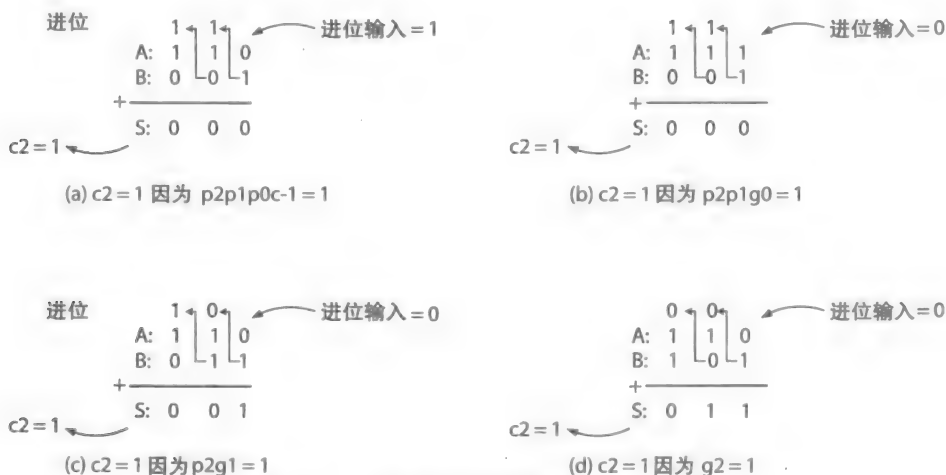


图 3-6 先行进位加法器概念的实例展示

当进位求出来以后，和值 s_0 、 s_1 和 s_2 也可以并行地生成，由如下等式确定。

$$s_0 = p_0 \oplus c_{-1}$$

$$s_1 = p_1 \oplus c_0$$

$$s_2 = p_2 \oplus c_1$$

图 3-7 展示了 3 位先行进位 (CLA) 加法器电路。 p 和 g 位用三个异或门和三个与门在 0.3ns 内并行生成，这里最大值 $\Delta \text{XOR} = 0.3\text{ns}$ 、 $\Delta \text{AND} = 0.2\text{ns}$ ，假设与非门有 0.1ns 延迟。这个电路也称为 PG 单元 (PGU)。 p 和 g 位和最初进位输入 c_{-1} 被送入进位生成单元 (CGU)。CGU 中的三个进位电路是独立的且可以在 0.2ns 内并行产生所有的进位 c_0 、 c_1 和 c_2 ，假设电路用与非门实现。使用另一套异或门，最后和值也可以用 p 和进位为输入并行生成。这些异或门可以组合成一个称为和单元 (SU) 的模块。

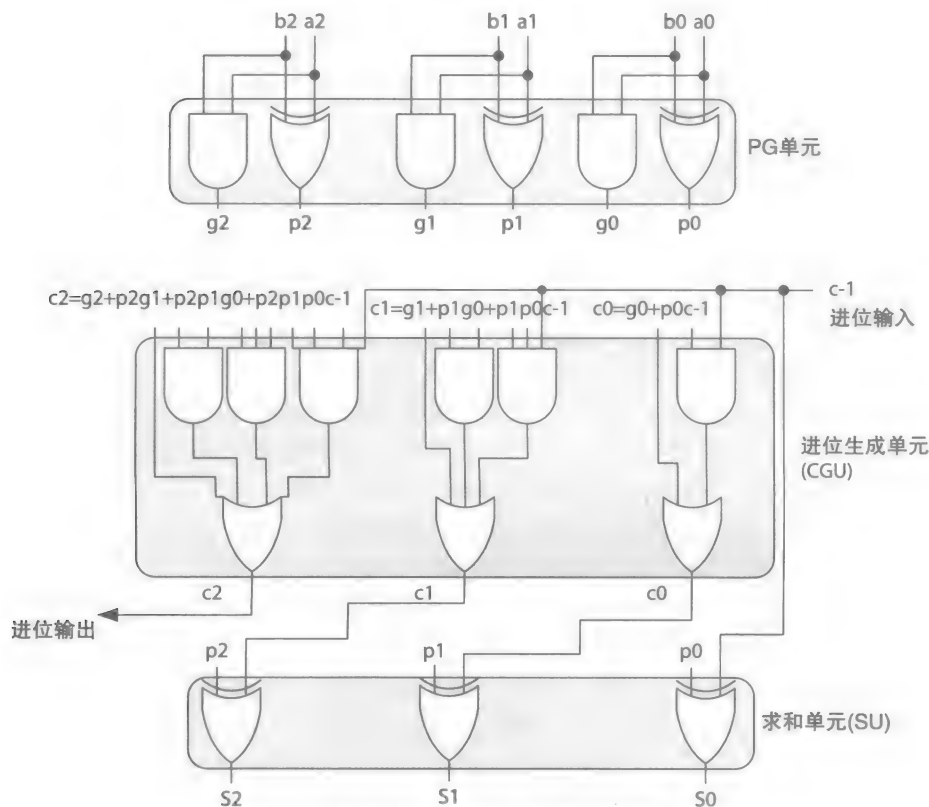


图 3-7 并行生成 c_0 、 c_1 和 c_2 进位的 3 位 CLA

有进位 $c_0 \sim c_7$ 的 8 位 CLA 仍然比较小，设计不会违反逻辑门扇入和扇出的限制。在这个例子中，进位 c_i 的电路用扇入 $\leq i + 2$ 的逻辑门实现。CLA (8) 的延迟为 0.8ns，如公式 (3-8) 所示，假设与非门有 0.1ns 延迟。CLA (8) 比 CPA (8) 运行速度要快两倍以上；然而，需要更多的硬件去实现其电路。

101

$$\begin{aligned} \Delta \text{CLA}(8) &= \Delta \text{PGU} + \Delta \text{CGU} + \Delta \text{SU} \\ &= 0.3\text{ns} + 0.2\text{ns} + 0.3\text{ns} \\ &= 0.8\text{ns} \end{aligned} \tag{3-8}$$

另一种 CLA 可以用与门实现 g 位、用或门实现 p 位及用 FA 输出和值 [1]。

1. 大型 CLA 加法器

对于很大的 n 值 (例如 $n > 8$)，将进位表达式连续代入后边的进位表达式中，如公式 (3-7) 中的进位 c_1 和 c_2 ，最后将产生一个非常长的进位表达式，这样对应的电路会需要很多有大量扇入数量的逻辑门。此外，一些逻辑门，比如产生 p 信号的异或门需要大量的扇出值，这样 p 才能作为多个 CGU 中的进位生成电路的输入去生成其他的进位信号，SU 中的和值电路也是如此。这样，当 n 是一个非常大的数 (例如 32 位或者 64 位) 时，需要另一种方法来限制每一个电路模块的扇入和扇出需求。

102

在这个实例中，进位表达式被分成不同的集合以便使在对应电路中的逻辑门的最大扇入和扇出可以控制在一个可接受的范围内。简单举例，当 $n = 8$ 时的情况如下。CLA (8) 的进位表达式分成三个集合，如公式 (3-9) 所示。每一个集合中的表达式都有少于三个逻辑项和少于三个的变量。则对应电路的扇入 ≤ 3 且扇出 ≤ 2 。例如，集合 1 中的 c_1 ，根据信号 p_0 ，

电路有最大扇入 = 3 和最大扇出 = 2。每一个集合中的进位表达式都是数据独立的；这样进位可以在输入可用时立即生成。对于集合 1 中的表达式， c_{-1} 是进位输入；在集合 2 中，进位输入为 c_2 ；在集合 3 中，进位输入为 c_5 。

集合 1:

$$c_0 = g_0 + p_0 c_{-1} \quad (\text{需要 } c_{-1} \text{ 作为进位输入})$$

$$c_1 = g_1 + p_1 g_0 + p_1 p_0 c_{-1} \quad (\text{需要 } c_{-1} \text{ 作为进位输入})$$

集合 2:

$$c_3 = g_3 + p_3 c_2 \quad (\text{需要 } c_2 \text{ 作为进位输入}) \quad (3-9)$$

$$c_4 = g_4 + p_4 g_3 + p_4 p_3 c_2 \quad (\text{需要 } c_2 \text{ 作为进位输入})$$

集合 3:

$$c_6 = g_6 + p_6 c_5 \quad (\text{需要 } c_5 \text{ 作为进位输入})$$

$$c_7 = g_7 + p_7 g_6 + p_7 p_6 c_5 \quad (\text{需要 } c_5 \text{ 作为进位输入})$$

在进位输入信号 c_{-1} 、 c_2 和 c_5 中，进位输入 c_{-1} 是最初的输入值，而其他两个进位输入必须由其他信号生成。它们由公式 (3-10) 决定并分为集合 4。

$$\begin{aligned} c_2 &= (g_2 + p_2 g_1 + p_2 p_1 g_0) + (p_2 p_1 p_0) c_{-1} \\ c_5 &= (g_5 + p_5 g_4 + p_5 p_4 g_3) + (p_5 p_4 p_3) c_2 \end{aligned} \quad (3-10)$$

使

$$g^*{}_0 = g_2 + p_2 g_1 + p_2 p_1 g_0$$

$$p^*{}_0 = p_2 p_1 p_0$$

$$g^*{}_1 = g_5 + p_5 g_4 + p_5 p_4 g_3$$

$$p^*{}_1 = p_5 p_4 p_3$$

因此，集合 4:

$$c_2 = g^*{}_0 + p^*{}_0 c_{-1} \quad (\text{需要进位输入 } c_{-1})$$

$$c_5 = g^*{}_1 + p^*{}_1 c_2$$

$$= g^*{}_1 + p^*{}_1 (g^*{}_0 + p^*{}_0 c_{-1})$$

$$= g^*{}_1 + p^*{}_1 g^*{}_0 + p^*{}_1 p^*{}_0 c_{-1} \quad (\text{此时也需要进位输入 } c_{-1})$$

103

考虑到当 c_2 和 c_5 表达式表示成 p^* 和 g^* 信号表达的形式时都是数据独立的。 p^* 和 g^* 信号也是数据独立的，且由 p 和 g 信号决定。也就是说， c_2 和 c_5 最终的表达式也和集合 1 ~ 集合 3 中的表达式类似，除了这些表达式要求以 p^* 和 g^* 信号作为输入。 p^* 中的 $*$ 号表示比特组中的进位传播。类似地， g^* 中的 $*$ 号表示比特组中的进位生成和传播。当 $p^* = 1$ 时，意味着 1 作为进位进入到块中将被传播为生成进位的 1。类似地，当 $g^* = 1$ 时，意味着在块中生成的进位 1 将被传输为进位输出。

集合 1 中的进位依赖于进位 c_{-1} 以及 p 和 g 信号；这样，它们可以在 p 和 g 信号生成之后立即并行地生成。考虑到因为所有 p^* 和 g^* 信号都可以在同一时间生成，和当 c_0 和 c_1 生成时，它们将被移动到集合 1 ~ 集合 3，如公式 (3-11) 所示：

集合 1:

$$c_0 = g_0 + p_0 c_{-1} \quad (\text{需要进位输入 } c_{-1})$$

$$c_1 = g_1 + p_1 g_0 + p_1 p_0 c_{-1} \quad (\text{需要进位输入 } c_{-1})$$

$$g^*{}_0 = g_2 + p_2 g_1 + p_2 p_1 g_0$$

$$p^*{}_0 = p_2 p_1 p_0$$

集合 2:

$$c_3 = g_3 + p_3 c_2 \quad (\text{需要 } c_2 \text{ 作为进位输入})$$

$c_4 = g_4 + p_4g_3 + p_4p_3c_2$ (需要 c_2 作为进位输入)

$g^*_1 = g_5 + p_5g_4 + p_5p_4g_3$

集合 3: $p^*_1 = p_5p_4p_3$ (需要 c_5 作为进位输入) (3-11)

$c_6 = g_6 + p_6c_5$ (需要 c_5 作为进位输入)

$c_7 = g_7 + p_7g_6 + p_7p_6c_5$ (需要 c_5 作为进位输入)

$g^*_2 = g_8 + p_8g_7 + p_8 + p_7g_6$ (当 $n > 8$ 时需要; 此时 $n = 8, g_8 = 0, p_8 = 0$)

$p^*_2 = p_8p_7p_6$ (当 $n > 8$ 时需要; 此时 $n = 8, p_8 = 0$)

8 位 CLA 的详细电路图如图 3-8 所示。所有 8 个进位 $c_0 \sim c_7$ 都在三步中产生，如下：

- 1) 集合 1 中的进位 c_0 和 c_1 和所有在集合 1、2 和 3 中的 p^* 和 g^* 信号首先生成；
- 2) 下一步，生成集合 4 中的进位 c_2 和 c_5 ；
- 3) 最后，生成集合 2 和集合 3 中的 c_3 、 c_4 、 c_6 和 c_7 。

104

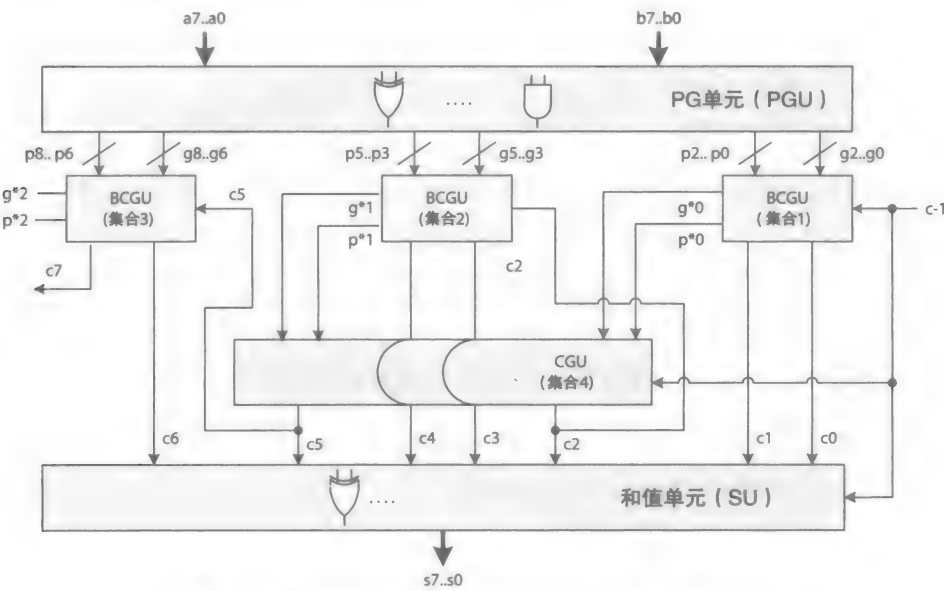


图 3-8 用 BCGU 实现的 8 位 CLA， p_8 和 g_8 设为 0

实现集合 1 ~ 集合 3 中每一个表达式的电路都称为块进位生成单元 (BCGU)。实现集合 4 中表达式的电路依旧为 CGU。当所有进位都生成之后， $c_{-1} \sim c_6$ 和 $p_0 \sim p_7$ 都被送进 SU 以便并行生成所有的和位值 $s_0 \sim s_7$ 。

对于很大值的 n (例如 $n = 32$ 或者 $n = 64$) 来说,CLA(n) 电路和图 3-8 中所示的 CLA(8) 电路是一样的。除了当 n 很大时，进位必须被分成更多的集合以避免 BCGU 或者 CGU 的扇入和扇出问题。以图 3-8 中的电路设计为基础，公式 (3-12) 估算了 CLA (n) 的传输延迟，假设与非门的延迟为 0.1ns。

$$\begin{aligned} \Delta \text{CLA}(n) &= \Delta \text{PGU} + \Delta \text{BCGU} + \Delta \text{CGU} + \Delta \text{BCGU} + \Delta \text{SU} \\ &= 0.3\text{ns} + 0.2\text{ns} + 0.2\text{ns} + 0.2\text{ns} + 0.3\text{ns} \\ &= 1.2\text{ns} \end{aligned} \tag{3-12}$$

大型 CLA 加法器比等价的 CPA 加法器要快得多，但是需要更多的硬件来实现。或者，我们可以设计一个一部分是 CLA 一部分是 CPA 的混合加法器。混合加法器比 CPA 要快，

但是比 CLA 要慢。例如, 16 位的混合加法器可以用两个 CLA (8) 片设计, 第一片中的进位 c_7 传送到第二片中。这个 16 位混合加法器要比 CPA (16) 快, 但是要比 CLA (16) 慢, 并且需要的硬件比 CLA (16) 要少一些。

2. HDL 模块

例 3-1 ~ 例 3-6 展示了用一个 PGU、三个 BCGU、一个 CGU 和一个 SU 设计的 8 位

105 CLA Verilog 描述。行为和结构模型都用于设计加法器。

例 3-1 8 位 CLA 加法器的结构化描述:

```
`include "pg_unit_8bits.v"
`include "carry_generate_8bits.v"
`include "sum_unit_8bits.v"

module adder_8bits
(
    input [7:0] a, b,
    input ci,
    output [7:0] s,
    output c6, c7
);
wire [7:0] c, p, g;
assign c6 = c[6];
assign c7 = c[7];
pg_unit_8bits          pgul(a, b, p, g);
carry_generate_8bits    cgul(p, g, ci, c);
sum_unit_8bits          sul(p, {c[6:0], ci}, s);
endmodule
```

例 3-2 用 BCGU 和 CGU 实现的 8 位进位生成模块的结构化描述:

```
`include "block_cla_carry_generate_3bits.v"
`include "cla_carry_generate_2bits.v"

module carry_generate_8bits (p, g, ci, c);
input [7:0] p, g;
input ci;
output [7:0] c;
wire [2:0] ps, gs;
block_cla_carry_generate_2bits bccg1(p[2:0], g[2:0], ci,
c[1:0], gs[0], ps[0]);
block_cla_carry_generate_2bits bccg2(p[5:3], g[5:3], c[2],
c[4:3], gs[1], ps[1]);
block_cla_carry_generate_2bits bccg3({1'b0, p[7:6]},
{1'b0, g[7:6]}, c[5], c[7:6], gs[2], ps[2]);
cla_carry_generate_2bits      ccg1(ps[1:0], gs[1:0], ci,
c[2], c[5]);
endmodule
```

例 3-3 3 位 BCGU 的行为描述:

```
module block_cla_carry_generate_2bits (p, g, ci, c, gs, ps);
input [2:0] p, g;
input ci;
output [1:0] c;
output gs, ps;
```

```
assign    c[0] = g[0] | p[0] & ci;
assign    c[1] = g[1] | p[1] & g[0] | p[1] & p[0] & ci;
assign    gs = g[2] | p[2] & g[1] | p[2] & p[1] & g[0];
assign    ps = p[2] & p[1] & p[0];
endmodule
```

例 3-4 2 位 CGU 的行为描述:

```
module cla_carry_generate_2bits (ps, gs, ci, c2, c5);
input [1:0] ps, gs;
input ci;
output c2, c5;

assign c2 = gs[0] | ps[0] & ci;
assign c5 = gs[1] | ps[1] & gs[0] | ps[1] & ps[0] & ci;
endmodule
```

例 3-5 8 位 PGU 的行为描述:

```
module pg_unit_8bits (a, b, p, g);
input [7:0] a, b;
output [7:0] p, g;

assign p = a ^ b;
assign g = a & b;
endmodule
```

例 3-6 n 位 SU 的行为描述:

```
module sum_unit_8bits (p, c, s);
input [7:0] p, c;
output [7:0] s;

assign s = p ^ c;
endmodule
```

3.4 减法器

减法器用图 3-9 所示 [2] 的一种方法生成两个 4 位输入 X 和 Y 的差值 D 。在方法 a 中，每次 $x_i < y_i$ 时，如果 $x_{i+1} > 0$ ，需要向 x_{i+1} (下一高位) 借 1，然后在 x_i 上加 2。如果 $x_{i+1} = 0$ ，那么当 $x_{i+2} > 0$ 时，就向 x_{i+2} 借 1，并把 2 加到 x_{i+1} 上。如果 $x_{i+2} = 0$ ，那么就向 x_{i+3} 借 1，把 2 加到 x_{i+2} 上，等等。这个过程递归一直到 $x_i \geq y_i$ ，且 $d_i = x_i - y_i$ 为 0 或 1。在图中， $x_0 = 0$ 小于 $y_0 = 1$ ；这样，需要从 x_1 上借 1。然而，因为 $x_1 = 0$ ，需要从 x_2 借 1。剩下的比特位也将进行相同的处理。此例不会产生借位输出，因为 $X = 12 = (1100)_2$ 大于 $Y = 3 = (0011)_2$ ，这样 $D = 12 - 3 = 9 = (1001)_2$ 。

[107]

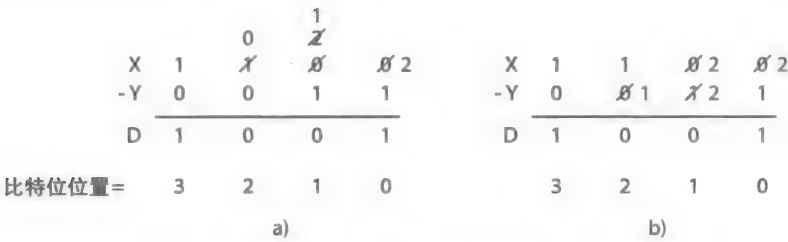


图 3-9 无符号减法过程: a) 借位方法; b) 抵扣法

在方法 b 中, 每一次 $x_i < y_i$ 时, 都会把抵扣 1 加到 y_{i+1} (下一高位) 上, 然后在 x_i 上加 2 生成 $d_i = x_i - y_i$ 为 0 或者 1。在例子中, $x_0 = 0$ 小于 $y_0 = 1$; 这样, 需要在 y_1 上加上 1, 让 $y_1 = 2$ 且 2 加到 x_0 上, 使得 $x_0 = 2$ 。这样 $d_0 = x_0 - y_0$ 为 1 ($2 - 1 = 1$)。下一步, $x_1 = 0$, 小于 $y_1 = 2$; 这样, 再次需要把 1 加到 y_2 上, 使其为 1, 且 2 加到 x_1 上, 使其为 2。结果为 $d_1 = 2 - 2 = 0$ 。剩下的比特位也按照相同的方法最终得到 $D = 9 = (1001)_2$ 。

n 位的借位传播减法器 (BPS) 类似于 n 位的 CPA, 用 n 个 1 位减法器片组合而成。每一片都从 X 输入一位数, 从 Y 输入一位数, 然后一个借位 / 抵扣位 (0 或 1) 和输出一位差值和下一借位 / 抵扣位, 如下第 i 位所示。 n 位 BSP 如图 3-10 所示。

$$\begin{array}{r}
 \dots \text{bi} \swarrow \text{bi-1} \swarrow \dots \\
 \dots \quad \quad \quad x_i \quad \quad \dots \\
 - \quad \dots \quad y_i \quad \dots \\
 \hline
 \dots \quad \quad di \quad \dots
 \end{array}
 \qquad
 \begin{array}{r}
 \dots 0 \swarrow 1 \swarrow \dots \\
 \dots \quad \quad \quad 0 \quad \quad \dots \\
 - \quad \dots \quad \quad 1 \quad \dots \\
 \hline
 \dots \quad \quad \quad 0 \quad \dots
 \end{array}$$

对于第 i 位的借位和抵扣位加法器算法将在后续讨论。除了圆括号用于强制优先, 所以展示了方法 a 的借位法和方法 b 的抵扣法概念, 差值为 $d[i]$ 的算式在两种算法中都是一样的。

减法算法

方法 a: 借位算法 (圆括号展示如何进行借位)

```

if x[i] > y[i]
    d[i] = (x[i] - b[i-1]) - y[i];      // borrow made to previous step is subtracted
                                        //from x
    b[i] = 0;
else if x[i] < y[i]
    d[i] = (x[i] + 2 - b[i-1]) - y[i];  // x borrows a 2 and borrow made to previous
                                        //step is subtracted from x
    b[i] = 1;                          // indicates the borrow from the next x bit
else if b[i-1] == 0 // x[i] = y[i]
    d[i] = x[i] - y[i];
    b[i] = 0;
else // b[i-1] = 1 and x[i] = y[i]
    d[i] = (x[i] + 2 - b[i-1]) - y[i];  // x borrows a 2 and borrow made to
                                        //previous step is subtracted from x
    b[i] = 1;                          // indicates the borrow from the next x bit
    
```

方法 b: 抵扣 (圆括号展示如何进行抵扣)

```

If x[i] > y[i]
    d[i] = x[i] - (b[i-1] + y[i]);      // the credit from the previous step is
                                        // added to y
    b[i] = 0;
else if x[i] < y[i]
    d[i] = (x[i] + 2) - (b[i-1] + y[i]); // x borrows a 2 and the credit from the
                                        // previous step is added to y
    b[i] = 1;                          // indicates a credit given to the next y bit
else if b[i-1] == 0 // x[i] = y[i]
    d[i] = x[i] - y[i];
    b[i] = 0;
else // b[i-1] = 1 and x[i] = y[i]
    d[i] = (x[i] + 2) - (b[i-1] + y[i]); // x borrows a 2 and the credit from the
                                        // previous step is added to y
    b[i] = 1;                          // indicates a credit given to the next y bit
    
```

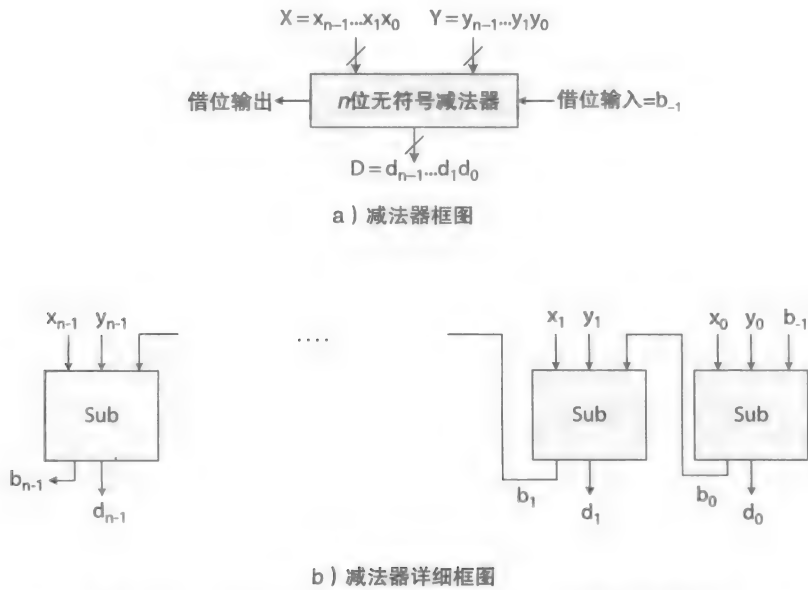


图 3-10 n 位 BPS: a) 减法器框图; b) 详细减法器框图

109

表 3-1 和公式 (3-13) 展示了 1 位减法器片的真值表和逻辑表达式。真值表简单地决定了之前讨论两种减法算法中的一种。考虑到差值和借位（抵扣位）表达式与 FA 的和值和进位表达式相似。或者，先行借位减法器 (BLA) 可以用 CLA 的方法设计。

$$d_i = x_i \oplus y_i \oplus b_{i-1}$$
$$b_i = (x_i \oplus y_i) b_{i-1} + \bar{x}_i y_i$$

(3-13)

3.5 2 的补码加法 / 减法器

对于带符号的算术运算，它们的数值可以表示成正数或者负数的 2 的补码。两个 n 位数字 $(A)_{2s}$ 和 $(B)_{2s}$ 的减法可以表示成 $(A)_{2s}$ 和 $(-B)_{2s}$ 的加法，如下所示，这里 $(B)_{1s}$ 表示简单地对 B 的每一位进行取反，即 B 的反码。

2 的补码减法算法

1) $(S)_{2s} = (A)_{2s} - (B)_{2s}$
$$= (A)_{2s} + [-(B)_{2s}]_{2s} \quad // + \text{表示“求和”而不是“或”}$$
$$= (A)_{2s} + [(B)_{1s} + 1]_{2s}$$
$$= [(A)_{2s} + (B)_{1s} + 1]_{2s}$$

2) 丢弃进位输出位

图 3-11 展示了 2 的补码减法算法过程。“+ 1” 用一位进位输入来实现。

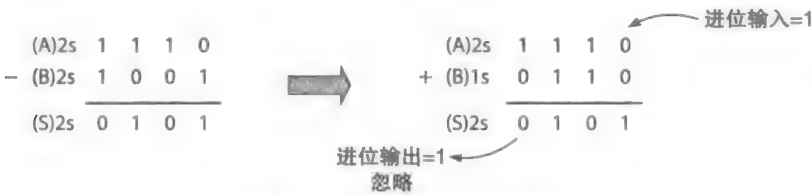


图 3-11 2 的补码减法过程

表 3-1 1 位减法器 (Sub) 真值表

| x_i | y_i | b_{i-1} | b_i | d_i |
|-------|-------|-----------|-------|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

类似地, 下列算法描述了 n 位 2 的补码加法。在这个例子中, 输入 B 没有变化且进位输入设为 0。

2 的补码加法算法

$$1) (S)_{2s} = (A)_{2s} + (B)_{2s} \\ = [(A)_{2s} + (B)_{2s} + 0]_{2s}$$

2) 丢弃进位输出位

2 的补码算术产生 n 位 2 的补码输出, 进位输出被忽略且不被计算到最后结果中。然而, 结果还是有可能溢出。例如, 考虑用 1 减最小的 n 位二进制补码负数, 或者 1 加上最大的 n 位 2 的补码正数。在两个例子中, 结果差值和和值都会超过 n 位负数和 n 位正数 2 的补码表示的范围。

当 A 和 B 都是 2 的补码正数时, 它们的符号位为 0。在这个例子中, 为了保证 $A + B$ 不会溢出, 加到符号位的进位必须为 0。这样会产生 0 作为进位输出; 不然就会产生溢出。例如, 考虑图 3-12 中的例子 a 和 c。在例子 a 中, $A = (0111)_{2s} = 7$ 且 $B = (0001)_{2s} = 1$ 都是 2 的补码正数, 此外, A 是最大的 4 位正数。如图中所示, 当 $B = 1$ 与 $A = 7$ 相加, 结果为 $(1000)_{2s} = -8$, 一个负数, 这就意味着溢出。在这个例子中, $c_3 = 1$, 且当其都与 0 的符号位相加的时候, 这就让符号位之和变为 1 (负数) 且 $c_4 = 0 \neq c_3 = 1$ 。要使溢出情况不会出现, 当两个 2 的补码正数相加时, c_3 和 c_4 都必须为 0。

在例子 c 中, 当两个负数 $A = -1$ 和 $B = -2$ 相加, 符号位都为 1, 如果 $c_3 = 1$ 产生 $c_4 = 1$ 则 $A + B$ 不会溢出。否则, 结果就会溢出, 生成一个正数为结果。从例子 a 和例子 c 中得到的结论是当 $c_3 = c_4$ 时, 结果不会溢出, 不论 A 和 B 是正数还是负数的 2 的补码形式。

例子 b 和 d 展示了减法。在这个例子中, 等式 $A - B$ 就是在计算 $A + (B)_{1s} + 1$ 。在例子 b 中, 当 $B = 1$ (正数) 去减 $A = -8$ 时 (最小的 4 位 2 的补码负数), c_3 为 0, 加到符号位 (都为 1) 上时, 产生 7 为结果 (不正确的结果)。注意, $c_4 = 1 \neq c_3 = 0$ 。在例子 d 中, $A = -1$ 和 $B = -2$ 都为 2 的补码负数, 且当相减时, 结果不会溢出。注意到在这个例子中, $c_3 = 1$ 与 $c_4 = 1$ ($c_3 = c_4$) 值相等。两个正数相减也不会产生溢出的情况。

表 3-2 为 n 位 2 的补码加法器 / 减法器溢出信号 ovf 的真值表。进位 c_{n-2} 与符号位相加产生最终进位 c_{n-1} 。公式 (3-14) 定义了溢出信号:

$$ovf = c_{n-1} \oplus c_{n-2} \quad (3-14)$$

图 3-13 展示了两个框图: a) 2 的补码加法器和 b) 2 的补码加法器。两个框图只有输入 $(B)_{2s}$ 和处理进位不相同。在图 3-13a 中, 输入 B 与 A 相加, 并没有改变表示形式。在图 3-13b 中, 输入 B 在与 A 相加之前, 经过了按位非运算。这样, 我们可以只有一个加法器模

块来组合两个电路框图实现一个加法器 / 减法器模块。可以用反相器模块来转换结果，当加法时输出 B ，当减法时输出 B 的非，当运算减法时， $E = e_{n-1} \cdots e_1 e_0$ 等于 B 的反码或者表示为 $(B)_{1s}$ 。模式信号 m 用来表示运算模式，当 $m = 0$ （进位输入为 0）时，运行加法，当 $m = 1$ （进位输入为 1），运行减法。

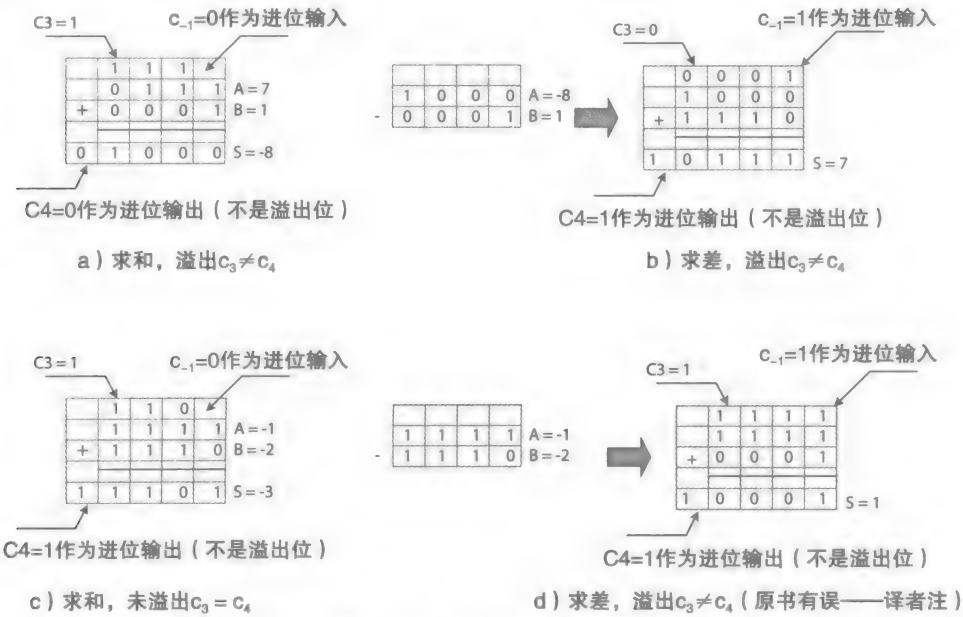


图 3-12 2 的补码运算实例

表 3-2 运算溢出信号的真值表

| $C_{(n-1)}$ | $C_{(n-2)}$ | ovf |
|-------------|-------------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

112

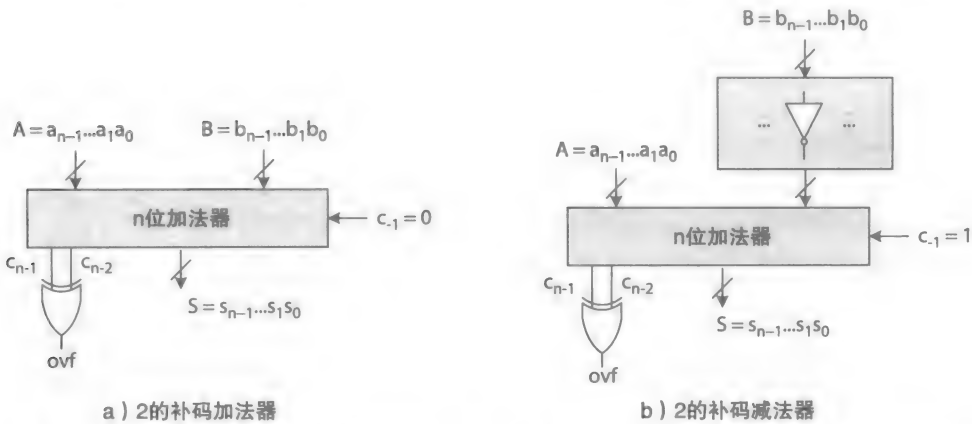


图 3-13 2 的补码加法器和减法器框图：a) 加法器框图；b) 包含加法器的减法器框图

表 3-3 为一位反相器片的真值表。最后组合设计如图 3-14 所示, 信号 m 也被用于最初的进位输入值。注意到有信号 m 的模块与进位输入连接不能用于这几位串行的大型加法器 / 减法器电路。如果需要设计位串行电路, 必须用分开的进位输入作为输入。

表 3-3 1 位反相器真值表

| | m | b_i | e_i |
|----------|-----|-------|-------|
| Add | 0 | 0 | 0 |
| Add | 0 | 1 | 1 |
| Subtract | 1 | 0 | 1 |
| Subtract | 1 | 1 | 0 |

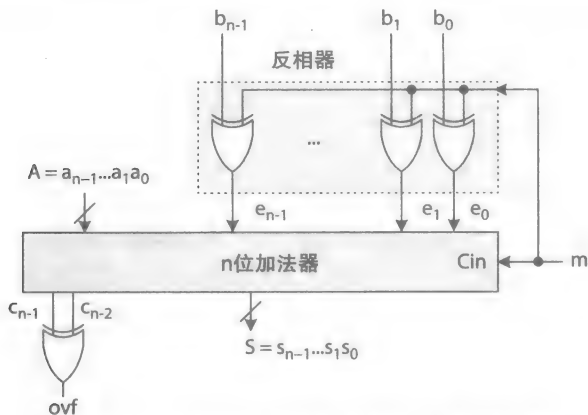


图 3-14 2 的补码加法器 / 减法器数据通路

113

3.6 算术逻辑单元

ALU 包含在所有的处理器中, 且不仅可以进行整数运算, 还可以进行位逻辑函数运算, 例如按位与、按位或。ALU 可以用在整数运算和逻辑指令的执行中。

例 3-7 设计一个 n 位、包含 7 种函数的 ALU, 使其可以完成加法、减法、增一、减一和按位与、按位或和按位非运算。3 位函数码 $F = f_2f_1f_0$ 用于选择 ALU 操作, 如表 3-4 所示。在运行算术操作 ($F = 0, 1, 2$ 或 3) 时, ALU 也可以输出溢出信号 ovf 。 $F = 7$ 不用于这个设计中, 当选择操作时, ALU 可以运行一个未知操作。稍后将讨论详细的位并行和位串行设计。

表 3-4 ALU 函数 $F = f_2f_1f_0$ 列表

| f_2 | f_1 | f_0 | 功 能 |
|-------|-------|-------|-----|
| 0 | 0 | 0 | 加法 |
| 0 | 0 | 1 | 减法 |
| 0 | 1 | 0 | 增一 |
| 0 | 1 | 1 | 减一 |
| 1 | 0 | 0 | 按位与 |
| 1 | 0 | 1 | 按位或 |
| 1 | 1 | 0 | 按位非 |
| 1 | 1 | 1 | 未定义 |

图 3-15 展示了 ALU 高层框图。当 F 指定了一个算术操作时，输入值 A 和 B 以及结果值 R 都表示为 2 的补码数。信号 ovf 为 1 时，表示运算有溢出。下一节将详细讨论位并行和位串行 ALU 的设计步骤。

3.6.1 设计部分：位并行

当 n 很大时，ALU 会被考虑为大型组合电路。用之前讨论的自顶向下位并行设计方法，ALU 的功能首先被分为算术和逻辑操作。4 个算术操作加法、减法、增一和减一被组合成 ALU 数据通路中的一个算术模块，如图 3-16 所示。三个按位逻辑操作将与门、或门和非门实现。8 位 4-1 的 MUX 选择输出 W 、 X 、 Y 和 Z 中的一个作为 ALU 的输出。

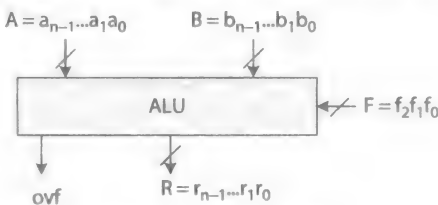


图 3-15 ALU 实例

114

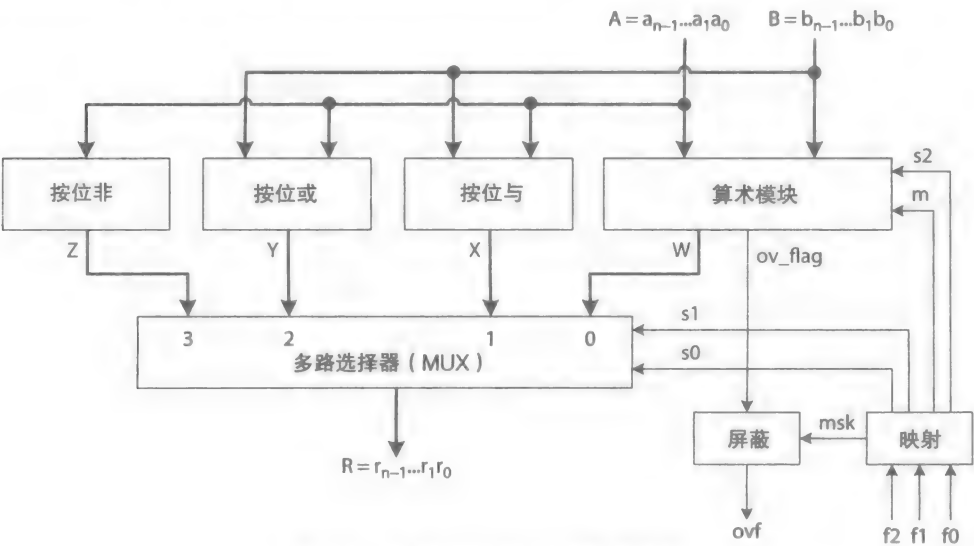


图 3-16 例 3-1 中的 ALU 数据通路

在 ALU 数据通路中， A 和 B 输入都与算术模块和三个位逻辑模块连接。这些模块同时对输入 A 和 B 进行运算并产生结果，但是只有一个被选择作为 ALU 的最终输出。这样，当 F 指定一个逻辑操作时，算术模块仍然可以根据 A 和 B 实时的值产生 ov_flag 信号值。然而，在某些实例中，溢出信号可以被屏蔽且不被 ALU 当作有效值输出。这个过程通过屏蔽模块完成。

数据通路还包含将 ALU 函数码 F 翻译成 ALU 数据通路中的中间信号 s_0 、 s_1 、 s_2 、 m 和 msk 的映射模块。当所有模块都被设计和连接完成之后，ALU 的设计过程就完成了。

数据通路中的 4-1 MUX 选择由算术和其他三个位逻辑模块的输出中的一个。当 F 指定一个逻辑操作 ($F = 4$ 到 6) 时，屏蔽模块将 ovf 置为 0 (无效)；否则， ovf 将会被置为 ov_flag 的值，作为算术模块的一个输出。

如果需要，自顶向下设计方法连续地应用到所有数据通路上的大型模块中，分为更小的电路模块，直到最底层电路模块足够小且只需要很少的输入实现为止。第 2 章中介绍的设计技术被用于设计每一个小型电路模块。位逻辑模块用 n 个 2 输入的与门、 n 个 2 输入的或门

115 和 n 个非门组成。这三个模块分别生成 n 位值 $X = x_{n-1} \cdots x_0$, $Y = y_{n-1} \cdots y_0$ 和 $Z = z_{n-1} \cdots z_0$, 如图所示, 这里第 i 位被如下式子定义:

$$x_i = a_i b_i$$

$$y_i = a_i + b_i$$

$$z_i = \overline{a_i}$$

作为一个例子, 图 3-17 展示了 4 位按位与逻辑的电路图。当 $A = a_3 a_2 a_1 a_0 = (1011)_2$ 且 $B = b_3 b_2 b_1 b_0 = (1101)_2$ 时, $X = x_3 x_2 x_1 x_0 = (1001)_2$ 由 $x_0 = a_0 \cdot b_0 = 1 \cdot 1 = 1, x_1 = a_1 \cdot b_1 = 0 \cdot 1 = 0$ 等。其他位逻辑模块也是类似的设计。

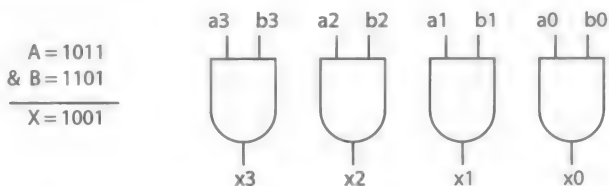


图 3-17 4 位按位与逻辑和其对应的逻辑电路

小型多路选择电路的设计在第 2 章中有过讨论。ALU 需要一个 n 位 4-1 的多路选择器, 可采用如下方法中的一种进行设计:

1) 用 n 个 1 位 4-1 MUX 设计

2) 用 n 位 2-1 MUX 设计。一个 n 位 2-1 MUX 用 n 个 1 位 2-1 MUX 设计, 如图 3-18 所示。

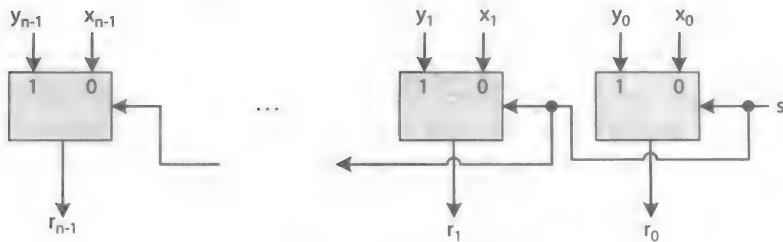
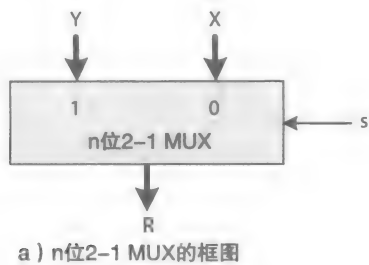


图 3-18 n 位 2-1 MUX: a) 框图; b) 1 位 2-1 MUX 片设计的框图

选择 2 具有优势, 其可以扩展到设计 n 位 $k-1$ MUX 且不用考虑任何扇入和扇出问题。表 3-5 展示了 n 位 4-1 MUX 的最小真值表。当其值为 0 的时候, 信号 s_1 选择 W 或 X 中的

一个，如果 s_1 值为 1，选择 Y 或 Z 中的一个。另一方面，当其值为 0 时，信号 s_0 选择 W 或 Y 中的一个，为 1 时，选择 Z 或 X 中的一个。这样，我们就可以将这个过程用于将 3 个 2-1 MUX 设计和实现 4-1 MUX，对于 n 位，如图 3-19 所示。例如，用 s_1s_0 表示数值为 2 的 2 位数时（即 $s_1 = 1$ 且 $s_0=0$ ），2-1 MUX 正确地选择了两个可能的输入候选 W 和 Y ，且最底端的 2-1 MUX 正确地选择了 Y 作为最后的输出。这个过程如图所示。

表 3-5 n 位 4-1 MUX 的化简真值表

| s_1 | s_0 | R |
|-------|-------|-----|
| 0 | 0 | W |
| 0 | 1 | X |
| 1 | 0 | Y |
| 1 | 1 | Z |

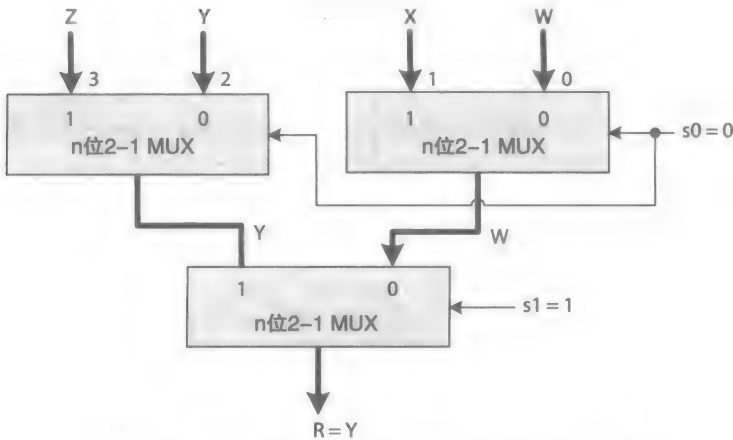


图 3-19 用 n 位 2-1 MUX 设计的 n 位 4-1 MUX

117

通常，先前的方法一共需要 $\log_2 k$ 层的 2-1 MUX 实现。公式（3-15）用于估计用 2-1 MUX 设计 K -1 MUX 的传输延迟。

$$\Delta_{k-1 \text{ MUX}} = \log_2 k * \Delta_{2-1 \text{ MUX}}$$

(3-15)

例如，8-1 MUX 需要三层 2-1 MUX，且 64-1 MUX 需要 6 层 2-1 MUX。如果使用不同的 MUX 组合，层的数量和总计延迟可以减少。例如，一个 8-1 MUX 也使用 2-1 MUX 和 4-1 MUX 的组合进行设计。在这个例子中，一个 4-1 MUX 可以设计成与或（SOP）或者或与（POS）电路以便使传输延迟最小。

算术模块的输入是两个 n 位 2 的补码数值 A 和 B 和两个控制信号 m 和 s_2 ，当 $n = 8$ 时如图 3-20 所示。算术模块有两个输出：一个 n 位 2 的补码 W 和一个（高电平）溢出信号 ov_flag 。表 3-6 列出了 m 和 s_2 与对应算术模块操作模式的值。如果其值为 0，信号 m 选择加法或者增一操作，如果值为 1，选择减法或者减一操作。 s_2 信号控制 2-1 MUX，当进行加法或者减法运算时，选择 B ，当进行增一或者减一运算时，选择 8 位数（00000001）₂。当 W 溢出时，信号 ov_flag 置为有效。

屏蔽模块的真值表和电路图如图 3-21 所示。当 $msk = 1$ 时，模块输出 $ovf = 0$ ，屏蔽掉从算术模块中产生的 ov_flag 信号；当 $msk = 0$ 时，输出 $ovf = ov_flag$ 。

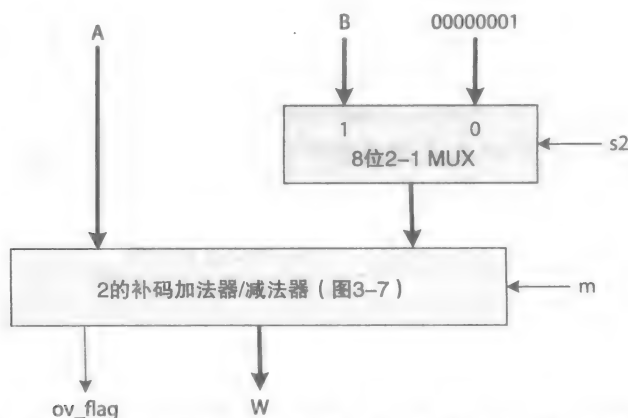
图 3-20 算术模块的详细框图；结果为 $A + B$ 、 $A - B$ 、 $A + 1$ 或者 $A - 1$

表 3-6 ALU 算术模块操作信号值

| | m | s_2 | 输出 |
|----|-----|-------|-------------|
| 增一 | 0 | 0 | $W = A + 1$ |
| 加法 | 0 | 1 | $W = A + B$ |
| 减一 | 1 | 0 | $W = A - 1$ |
| 减法 | 1 | 1 | $W = A - B$ |

| msk | ov_flag | ovf |
|-------|------------|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

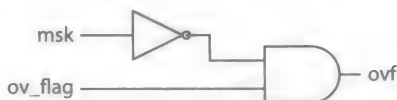


图 3-21 溢出信号屏蔽模块：对应真值表及电路图

表 3-7 展示了映射模块的真值表。表中的控制信号，除了 msk ，可以视为无关项 (d)。例如，当 F 指定一个逻辑操作（例如 $F = 4 \sim 6$ ）时，算术模块的输出 W 不会被 MUX 选中作为输出。然而，信号 ov_flag 必须屏蔽，使得 ALU 不输出 ovf 信号。映射模块的真值表如图 3-22 所示。

此外，因为 $F = 7$ 没有定义，所以所有控制信号，除了 msk ，在 $F = 7$ 时都可以视为无关项。然而，因为图 3-22 中的映射电路生成 $s_2 = 0$ ， $s_1 = 1$ ， $s_0 = 0$ ， $m = 1$ ，且当 $F = 1$ 时， $msk = 1$ ，这些信号值对应到按位或操作，ALU 会在当 $F = 5$ 或 7 时进行按位或操作。

表 3-7 ALU 映射模块真值表

| | f_2 | f_1 | f_0 | s_2 | s_1 | s_0 | m | msk |
|-----|-------|-------|-------|-------|-------|-------|-----|-------|
| 加法 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 减法 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 增一 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 减一 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 按位与 | 1 | 0 | 0 | d | 0 | 1 | d | 1 |
| 按位或 | 1 | 0 | 1 | d | 1 | 0 | d | 1 |
| 按位排 | 1 | 1 | 0 | d | 1 | 1 | d | 1 |
| 未定义 | 1 | 1 | 1 | d | d | d | d | 1 |

(续)

| f_2 | f_1 | f_0 | a | b | ci | co | r | 功能 |
|-------|-------|-------|---|---|----|----|---|-----|
| | | | 0 | 1 | 1 | 1 | 0 | 加法 |
| | | | 1 | 0 | 0 | 0 | 1 | |
| | | | 1 | 0 | 1 | 1 | 0 | |
| | | | 1 | 1 | 0 | 1 | 0 | |
| | | | 1 | 1 | 1 | 1 | 1 | |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 减法 |
| | | | 0 | 0 | 1 | 1 | 1 | |
| | | | 0 | 1 | 0 | 1 | 1 | |
| | | | 0 | 1 | 1 | 1 | 0 | |
| | | | 1 | 0 | 0 | 0 | 1 | |
| | | | 1 | 0 | 1 | 0 | 0 | |
| | | | 1 | 1 | 0 | 0 | 0 | |
| | | | 1 | 1 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | d | 1 | 0 | 1 | 增一 |
| | | | 1 | d | 0 | 0 | 1 | |
| | | | 1 | d | 1 | 1 | 0 | |
| 0 | 1 | 1 | 0 | d | 1 | 1 | 1 | 减一 |
| | | | 1 | d | 0 | 0 | 1 | |
| 1 | 0 | 0 | 1 | 1 | d | d | 1 | 按位与 |
| 1 | 0 | 1 | 0 | 1 | d | d | 1 | 按位或 |
| | | | 1 | 0 | d | d | 1 | |
| | | | 1 | 1 | d | d | 1 | |
| 1 | 1 | 0 | 0 | d | d | d | 1 | 按位非 |
| | | | 1 | d | d | d | 0 | |
| 1 | 1 | 1 | d | d | d | d | d | 未定义 |

因为当 $F = 7$ 时,ALU 不会进行任何操作,所以对应的表项被设为无关项。真值表太大,所以不能手动的方式进行化简。用 Espresso 方法输出的质主蕴含将稍后列出。虽然没有特定的函数去定义当 $F = 7$ 时的情况,当 $F = 7$ 时,1 位 ALU 片输出为 1。注意到输出 0xFF 也能被 8 位 2 的补码表示为 -1。表 3-9 展示了当 $F = 7$ 输出为 -1 时的位串行 ALU 操作的最终列表。

[120]

表 3-9 用 1 位 ALU 片组成的 n 位位串行 ALU 最终 ALU 功能列表

| f_2 | f_1 | f_0 | 功能 |
|-------|-------|-------|-----|
| 0 | 0 | 0 | 加法 |
| 0 | 0 | 1 | 减法 |
| 0 | 1 | 0 | 增一 |
| 0 | 1 | 1 | 减一 |
| 1 | 0 | 0 | 按位与 |
| 1 | 0 | 1 | 按位或 |
| 1 | 1 | 0 | 按位非 |
| 1 | 1 | 1 | -1 |

1 位 ALU 片的质主蕴含：

```
#1-bit ALU-slice: Add, Sub, Inc, Dec, And, Or, and Not
#Input signal labels
#output bit label
#list of min-terms
.i 6
.o 2
.ilb f2 f1 f0 a b ci
.ob co r
.p 15
00-010 01
0-01-1 10
-0011- 10
0-10-1 10
-0101- 10
0--100 01
01-1-0 01
00--11 10
0--001 01
10-11- 01
-0-111 01
11-0-- 01
-1-0-1 01
1-1-1- 01
1-11-- 01
.e
```

121

通常当字长不是标准字长的时候，位串行设计是有好处的（例如，位串行加密硬件所用的 256 位或 1024 位操作数），或者其等价的位并行设计需要更多的硬件去实现相同的逻辑。

3.7 设计实例

作为之前讨论的实现整数加法和减法操作组合电路的补充，以下小节将介绍组合整数乘法和除法电路。乘法的基本操作为加法，除法的基本操作为减法。然而，一些乘法和除法算法会同时用到加法和减法。同时使用加法和减法操作的 2 的补码乘法器将在第 6 章进行介绍。

122

3.7.1 乘法器

图 3-24 展示了 4 位无符号乘数 $B = b_3b_2b_1b_0$ 和 4 位无符号被乘数 $A = a_3a_2a_1a_0$ 的乘法过程。乘法过程的每一步产生一个加数。在图中， $(1001)_2$ 、 $(1001)_2$ 、 $(0000)_2$ 和 $(1001)_2$ 是产生的 4 个加数，分别由 b_0 、 b_1 、 b_2 和 b_3 和 A 相与产生的。每一个新产生的加数会左移 $k - 1$ 次， k 代表乘法的步数。如图所示，4 个加数在相加产生最终的乘积 P 之前都被分别按顺序左移 0、1、2 和 3 位。

1001A

*1011B

1001

1001

0000

+1001

1100011P=B*A

图 3-24 4 位无符号二进制乘法实例

例如图 3-25 所示的 $n = 4$ 乘法过程，设计组合 n 位乘法器电路的一种方法是用 $n - 1$ 个 n 位加法器和 n 个 n 位按位与模块组成。这个设计很直观且是基于图 3-24 所示的步骤进行的。但是这种设计有较长的传输延迟因为除了前两个加数，其他的加数每次都只加一个，这样会造成很长的从输入到输出的信号路径。

或者我们可以使用全加器片来对加数进行加法操作，每一次操作一位。当全加器连接在一起时，就产生了一种叫作阵列乘法器的二维结构。图 3-26 展示了一个用 6 列全加器实现的 4 位阵列乘法器。加数的和值每一次都能够确定一位，有点类似于多个加数的手工加法。在图中，一个加数用其独立的位，如 $a_i b_j$ 表示，这里 a_i 表示被乘数 A 的第 i 位， b_j 表示乘数 B 的第 j 位。乘积的每一位都是一位由全加器链在列中产生的最终和。在每一列中，没有用到的输入将连接到 0 上。最后一位乘积位 p_7 等于全加器列的进位输出位。

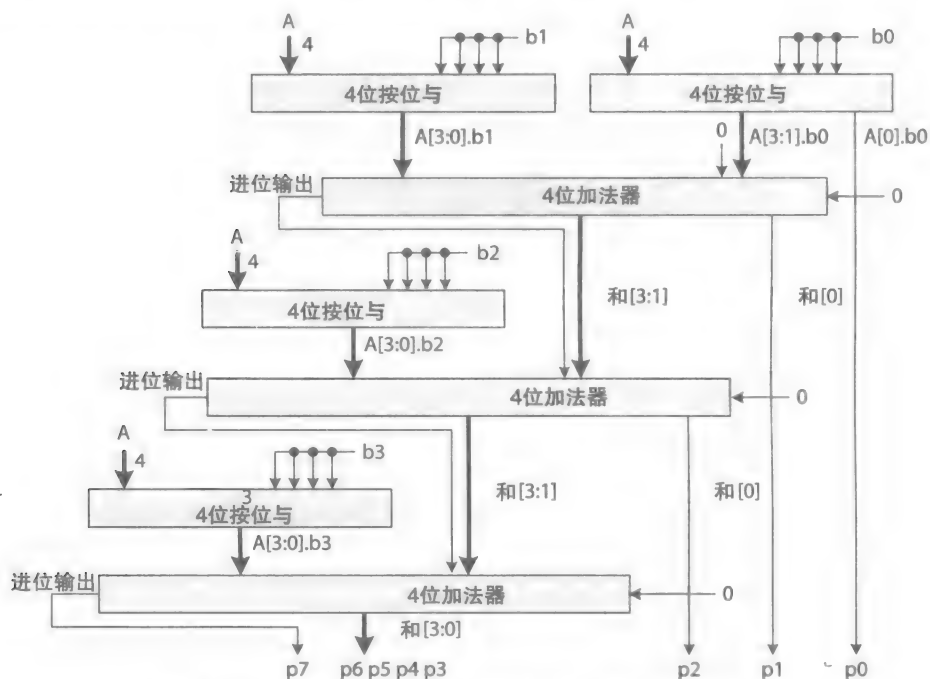


图 3-25 用 n 位加法器模块组成的 4 位无符号乘法器

在图中，最后一行的全加器组成了一个 CPA，CPA 可以用 CLA 加法器代替从而将乘法器的总传输延迟降到最小。

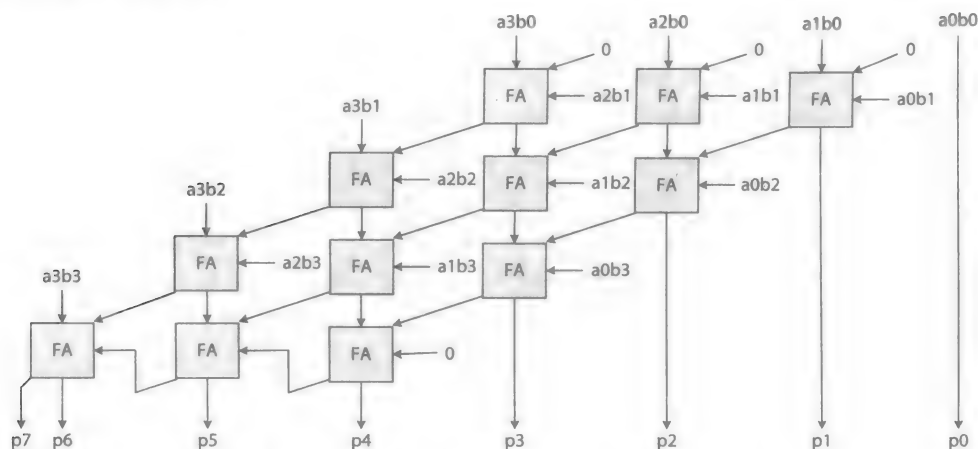


图 3-26 用阵列全加器实现的 4 位阵列乘法器

3.7.2 除法器

图 3-27 展示了用一个 4 位无符号被除数 (分子) $N = n_3n_2n_1n_0 = 4'b1011$ 除以 4 位除数 (分母) $D = 4'b0010$ 产生 4 位商 $Q = q_3q_2q_1q_0 = 4'b0101$ 和 4 位余数 $R = 4'b0001$ 的除法过程。 N 被从左填充 $n - 1$ 个 0, 在这个例子中为 3 个 0; 即最开始的被除数为 $\{000, N\}$, 这里的 $\{\}$ 用于表明为串联。在每一个步骤中, 除数 D 将被被除数较高的 n 位减去, 在除法的第 k 步中定义为 A_k 。如果 $D \leq A_k$, 对应的商位为 1; 否则商位为 0。

图中所示的除法步骤也被称为恢复除法算法, 因为每一次当 $A_k < D$ 时 (例如, $A_3 = 4'b0001 < D = 4'b0010$), $A_k - D < 0$, 这样 A_k , 不是余数 $R_k = A_k - D$ 用于下一步除法的开始, 即为“恢复”。在这个例子中, A_k 较低的 $n - 1$ 位与 N 的下一位连接组成下一个 n 位被除数 A_{k-1} 。需要特别指出的是, 对于 $n = 4$, 如图所示的 $\{000, N\}$ 除以 D 的除法步骤和产生 4 位 Q 和 4 位 R 的步骤如下所示:

- 1) $R_3 = A_3 - D$ ($4'b0001 - 4'b0010 = 4'b1111$) 产生 $R_3 = -1$, 且借位输出为 1 (例如 $bo_3 = 1$)、 $q_3 = 0$ 。
- 2) $R_2 = A_2 - D$ ($4'b0010 - 4'b0010$) 产生 $R_2 = 0$, 且 $bo_2 = 0$ 、 $q_2 = 1$ 。
- 3) $R_1 = A_1 - D$ ($4'b0001 - 4'b0010$) 产生 $R_1 = -1$, 且 $bo_1 = 1$ 、 $q_1 = 0$ 。
- 4) $R_0 = A_0 - D$ ($4'b0011 - 4'b0010$) 产生 $R_0 = 1$, 且 $bo_0 = 0$ 、 $q_0 = 1$ 。最终的余数 $R_0 = 4'b0001$ 。

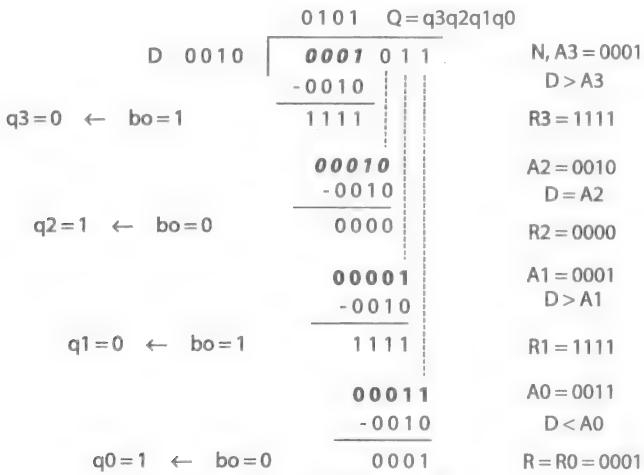


图 3-27 恢复除法实例

图 3-28 展示了 4 位位并行恢复除法的数据通路。在除法的每一步中都需要用到一个减法器、一个非门和一个 MUX。减法器计算 $R_k = A_k - D$ 并产生 bo_k 。非门产生商位, 例如 $q_k = \overline{bo_k}$ 。MUX 用于用 q_k 位选择 A_k 或者 R_k 。公式 (3-16) 估计了每一步除法的传输延迟。

$$\Delta_{\text{division-step}} = \Delta_{\text{subtractor}} + \Delta_{\text{MUX}}$$

(3-16)

阵列除法器, 类似于一个阵列乘法器, 可以用一个 1 位除法片阵列来组成。每一片将完成一个组合减法器 MUX 的功能。组合起来的功能可以翻译为最小 SOP 或者 POS 表达式的真值表 (更多细节参照练习部分)。

对于很大的 n , 一个算术功能设计为一个组合电路需要相当多的逻辑门来实现。当算

法是重复的并且可以替代地重复实施时更是如此，例如乘法器和除法器。例如，除了用 4 个减法器、4 个 MUX 和 4 个非门来实现如图 3-28 所示的 4 位恢复除法器，我们也可以用一个减法器、一个 MUX、一个非门和一系列的寄存器来产生 4 步除法中的 4 个商值。每一步除法的结果都存在寄存器中。然而，一个硬件模块重复使用需要一些额外的硬件来控制每一步的时序，且会稍微增加获得最后结果的总时间。这个设计将在第 6 章中进行讨论。

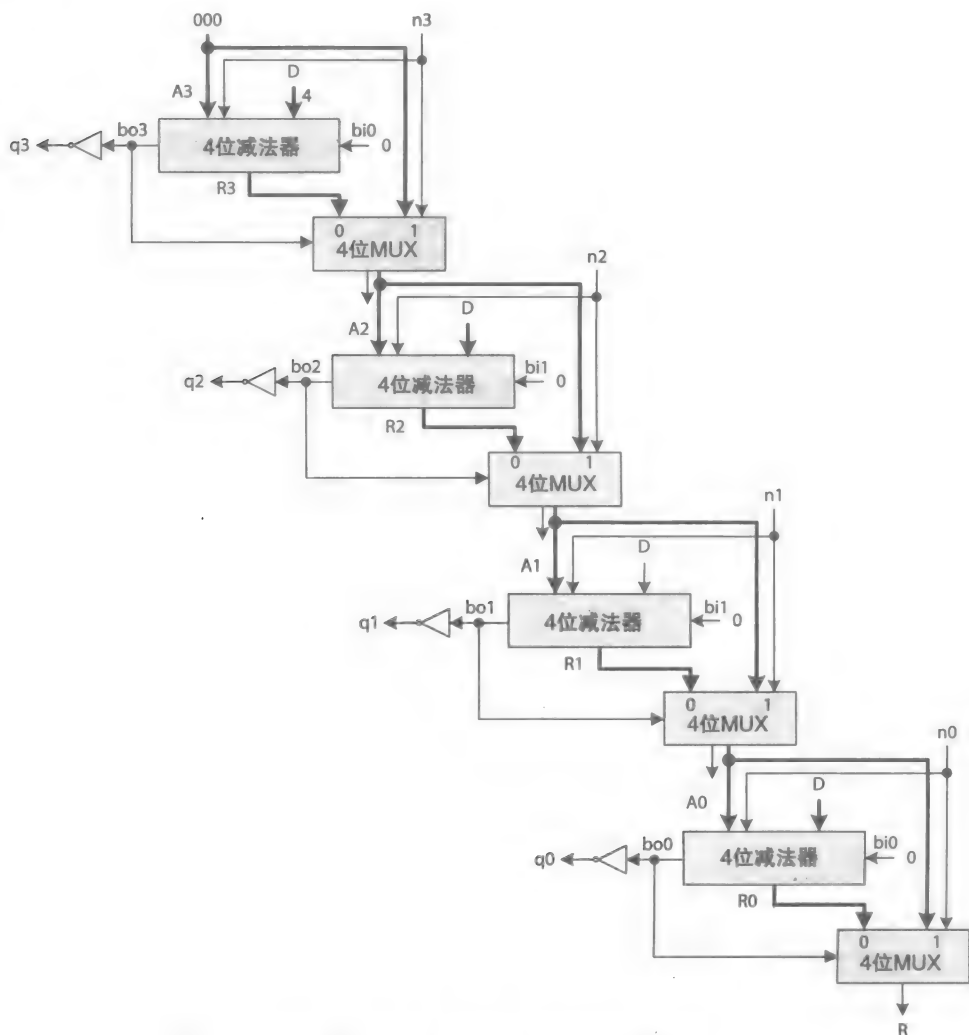


图 3-28 4 位“恢复”除法器数据通路

3.8 实数算术

在第 1 章中已经简要地讨论过实数，例如，浮点数的表示方法。表 3-10 作为一个例子展示了 3 位指数值的 3 种不同表示方法，分别为 2 的补码有符号数、偏移量为 3 的偏置数和偏移量为 4 的偏置数。这三种指数表示法的指数范围分别为 $-4 \sim +3$ 、偏移量为 3 时的 $-3 \sim +4$ 和偏移量为 4 时的 $-4 \sim +3$ 。

表 3-10 3 位带符号值和偏置指数值列表

| 3 位指数 | 2 的补码数 | 偏置指数, 偏移量 = 3 | 偏置指数, 偏移量 = 4 |
|-------|--------|---------------|---------------|
| 000 | 0 | $-3 = 0 - 3$ | $-4 = 0 - 4$ |
| 001 | 1 | $-2 = 1 - 3$ | $-3 = 1 - 4$ |
| 010 | 2 | $-1 = 2 - 3$ | $-2 = 2 - 4$ |
| 011 | 3 | $0 = 3 - 3$ | $-1 = 3 - 4$ |
| 100 | -4 | $1 = 4 - 3$ | $0 = 4 - 4$ |
| 101 | -3 | $2 = 5 - 3$ | $1 = 5 - 4$ |
| 110 | -2 | $3 = 6 - 3$ | $2 = 6 - 4$ |
| 111 | -1 | $4 = 7 - 3$ | $3 = 7 - 4$ |

通常情况下，使用偏置指数能让设计者有更多的自由来决定计算机系统中实数的集合。在这个例子中，当偏移量为 3 时，最大正指数为 4，而当偏移量为 4 时，最大正指数为 3。类似地，当偏移量为 3 时，最小负指数为 -3，当偏移量为 4 时，最小负指数为 -4。这表明了当偏移量为 3 时，将有更多可以表示为浮点数的实数 $> |1|$ (绝对值)，但当偏移量为 4 时，将有更多可以表示为浮点数的实数 $< |1|$ (绝对值)。

3.8.1 浮点数标准

IEEE 754 标准 [3] 包含了三种浮点数表示方法，分别为单浮点数、双浮点数和扩展双浮点数。表 3-11 列出了每一种表示方法的指数和分数范围。单浮点数和双浮点数在存储器或者外存中尾数分别是 23 和 52 位，在寄存器或者内存中，尾数分别为 24 和 53 位。扩展双浮点数表示方法有 64 位尾数且用于增加浮点数算法的精确度。尾数一般用带符号的量级数字表示，拥有独立的符号位。在存储器或者外存中没有扩展双浮点数的表示方法。

表 3-11 IEEE 754 浮点数标准

| 形 式 | 尾数符号位 | 偏置指数 | 外部尾数 | 内部尾数 | 总大小 (内部) |
|--------|-------|------|------|------|----------|
| 单浮点数 | 1 位 | 8 位 | 23 位 | 24 位 | 32 位 |
| 双浮点数 | 1 位 | 11 位 | 52 位 | 53 位 | 64 位 |
| 扩展双浮点数 | 1 位 | 15 位 | — | 64 位 | 80 位 |

* 奔腾处理器系列中的实现

此外，IEEE 标准将浮点数分为 5 个数据类，单数据类列出如下：

- 0
- 23 位非规格化尾数
- 24 位规格化尾数 (只有 23 位可以储存在存储器中)
- 无穷大
- NaN (不是一个数)，表示一个非法的浮点数或者操作

公式 (3-17) 表示了非偏置指数 e 和偏置指数 E 之间的关系。单浮点数或者双浮点数的表示形式为 $1.F \times 2^E$ ，在小数点前有一个固定的 1；然而，这个 1 没有被存储在存储器里。 F 是一个外部 (存储器) 尾数，而 $1.F$ 是一个内部 (寄存器) 尾数。一个非规格化浮点数被定

义为 $0.F \times 2^E$ ，有一个固定的 0 在小数点前。

$$\begin{aligned} E &= e + \text{偏移量} \\ e &= E - \text{偏移量} \end{aligned} \tag{3-17}$$

表 3-12 展示了每一种数据类的表示范围。在表中， e_{\min} 和 e_{\max} 表示规格化浮点数的真正（非偏移量）指数范围。

例如，假设偏移量为 3 的 3 位偏置指数，公式（3-18）定义了每一种数据类的 E 和 F 值。

$$\begin{aligned} \text{零:} \quad & E = 0, F = 0 \\ \text{非规格化:} \quad & E = 0, F > 0 \\ \text{规格化:} \quad & 1 \leq E \leq 6, F \geq 0 \\ \text{无穷大:} \quad & E = 7, F = 0 \\ \text{非法值:} \quad & E = 7, F > 0 \end{aligned} \tag{3-18}$$

表 3-12 IEEE 浮点数据类

| 非偏置指数 | 外部尾数 | 数据类范围 | 存储器中表示 { 符号, E, F } |
|---------------------------------|------------|-----------------------------|---------------------|
| $e = e_{\min} - 1$ | $F = 0$ | $+/- 0$ (零) | {0/1, 0, 0} |
| $e = e_{\min} - 1$ | $F > 0$ | $+/- 0.F \times 2^e$ (非规格化) | {0/1, 0, F} |
| $e_{\min} \leq e \leq e_{\max}$ | $F \geq 0$ | $+/- 1.F \times 2^e$ (规格化) | {0/1, E, F} |
| $e = e_{\max} + 1$ | $F = 0$ | $+/- \infty$ (无穷大) | {0/1, E, 0} |
| $e = e_{\max} + 1$ | $F > 0$ | $+/- \text{NaN}$ (非数值) | {0/1, E, F} |

{} 表示级联

128

3.8.2 浮点数据空间

一个浮点数据空间可以用计算机系统中浮点数能表示的实数范围表示。图 3-29 展示了用正实轴表示的浮点数据空间。0 和 $2^{e_{\min}}$ （不包括 0 和 $2^{e_{\min}}$ ）之间的水平虚线表示非规格化数据空间。细线和粗线表示当尾数只有 2 位时用浮点数表示的特殊实数。在表示法中浮点数更多的位数意味着可以表示更多的实数。

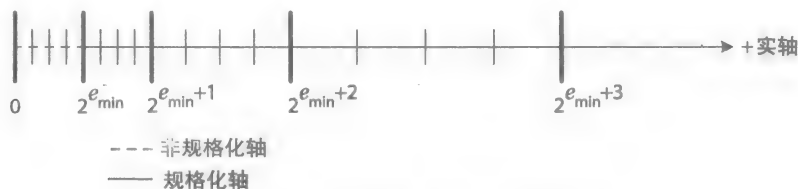
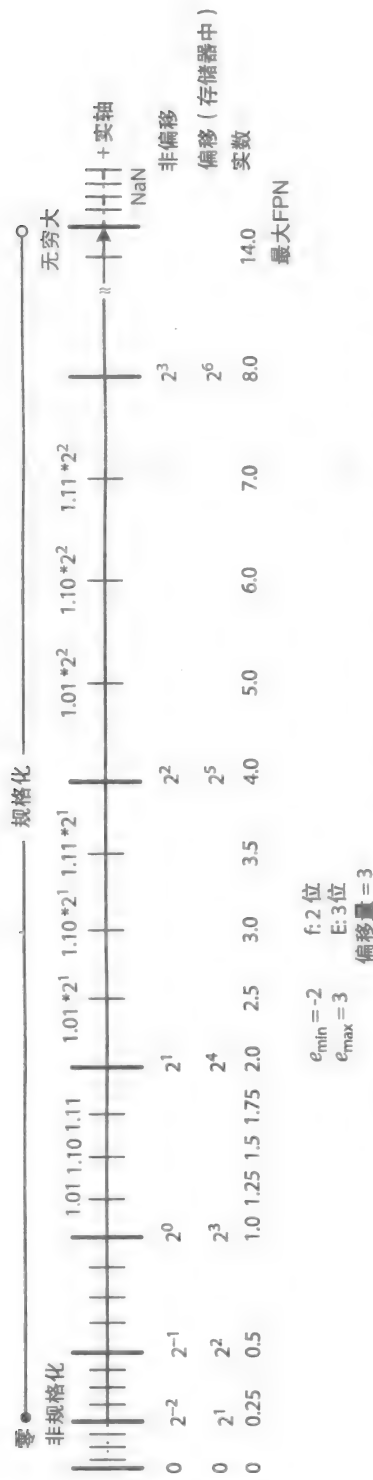


图 3-29 2 位尾数的浮点数据空间

例如，在单浮点表示法中，偏移量为 127 时，8 位非偏置指数的范围是从 $e_{\min} = -126$ 到 $e_{\max} = 127$ 。每一个尾数为 23 位。其数据空间在每一对粗线之间有 $2^{22} - 1$ 条细线。双浮点数据空间在每一对粗线中有 $2^{52} - 1$ 条细线。

图 3-30 展示了可以表示为有 1 位符号位，2 位尾数和偏移量为 3 的 3 位指数的 6 位浮点数的正实数。如图所示，17 个实数范围从 0.25 ~ 14.0（包括）可以表示为 6 位浮点数形式。从 0 ~ 0.25（不包括）的浮点数会被认为是非规格化的。

129



例 3-8 给出实数 +10.75 的对应的外部（存储器）单浮点数表示形式。

解：首先，将数值转换成二进制形式。然后将其表示形式转换为科学表示形式。

$$\begin{aligned}
 +10.75 &= (1010.11)_2 \\
 &= (1010.11)_2 * 2^0 \\
 &= (1.01011)_2 * 2^3 \quad (1.F \times 2^e \text{ 的非偏移科学形式}) \\
 &= (1.01011)_2 * 2^{3+127} \quad (\text{偏移量} = 127 \text{ 的偏移科学形式}) \\
 &= (1.01011)_2 * 2^{130} \quad (1.F \times 2^E \text{ 的 IEEE 形式})
 \end{aligned}$$

结果为如下的 32 位数，由一位符号位 = 0, 8 位偏置指数 $E = 130 = (10000010)_2$ 和 23 位尾数 = $(010110\cdots 0)_2$ 表示的数值：

| | | |
|---|----------|--------------------------|
| 0 | 10000010 | 010110000000000000000000 |
|---|----------|--------------------------|

或者写成十六进制：0x412C0000。1.F 中的 1 没有被存储至存储器中。

除了符号位，浮点数的正数和负数的表示方法是一样的。负浮点数的符号位为 1。例如，-10.75 在存储器中单浮点数表示为 0xC12C0000。

二维展示

表示浮点数据空间的另一种方法是用矩形域表示 [4]，如图 3-31 所示，表示的是单浮点数。两个分开的二维表示用于分别表示正浮点数和负浮点数的数据空间。在图中， x 轴上的任意点表示指数值，而 y 轴上的点表示尾数值。两轴的数据范围不相同。

和图 3-29 中的一维表示相比，二维表示更容易定位一个特定浮点数的位置，或者一个浮点函数的定义域和值域，例如余弦值。例如，余弦函数的值域范围是 0 ~ 1.0 之间包括 0 和 1.0 所有可表示的浮点值。例如，函数的定义域和值域可能用于生成 FPU 的测试向量 [4]。图 3-32 展示了在二维浮点数据空间中一些示例浮点值的位置。

最小的正规格化浮点值在图中被标记为第一项且位置在正规格化空间的左下角，其 $E = 1$ 且 $F = 0$ 。最小的（例如最大级）负规格化浮点数被标记为第二项且位置在负规格化空间的右上角，其 $E = 254$ 且 F 为全 1，即 255。 $+1.0$ 被标记为第三项且其位置在 + 规格化空间的最下方，其 $E = 127$ 且 $F = 0$ 。 \pm 零其 $E = 0$ 且 $F = 0$ 。 \pm 无穷大其 $E = 255$ 且 $F = 0$ 。而且在 Intel 奔腾处理器系列中，有一些其他特殊的浮点数，例如 NaN (QNaN) 表示一个非法操作数——例如，-1 的开方——或者信号 NaN (SNAN) 表示一个非法操作 [5]。

双浮点数据空间的二维展示也是类似的，除了用于表示指数和尾数数字的位数不一样以外，在双浮点数据空间中，指数为 11 位，尾数为 52 位。

3.8.3 浮点运算

一个浮点数的指数和尾数分别都是整数且在浮点运算中可以分别操作。例如，两个实数 $0.1075 (10.75 \times 10^{-2})$ 和 $72.5 (0.725 \times 10^2)$ 进行相加，更小的尾数将右移小数点，使得小数点对齐，进而得出两数之和 $72.5075 (0.725075 \times 10^2)$ 。较小尾数移动的位数取决于两个指数。两个二进制实数相加的算法和十进制是相同的，将在以下列出。标记 $A.s$ 和 $B.s$, $A.E$ 和 $B.E$, $A.F$ 和 $B.F$ 分别表示两个浮点数 A 和 B 的符号位、偏置指数值和外部（存储器中）尾数值，即

$$\begin{aligned}
 A &= \{A.s, A.E, A.F\} \\
 B &= \{B.s, B.E, B.F\}
 \end{aligned}$$

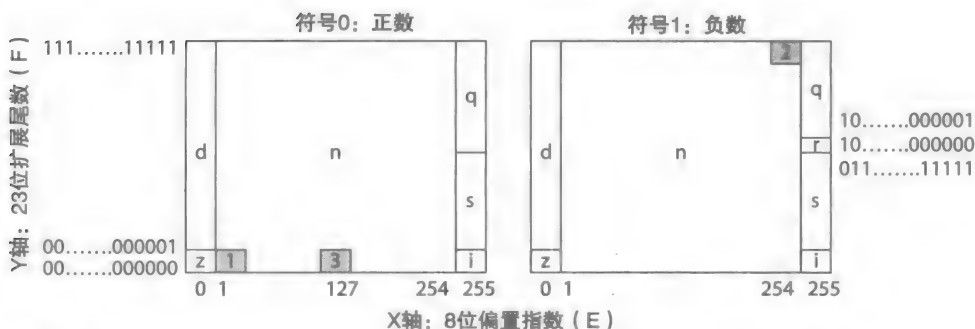
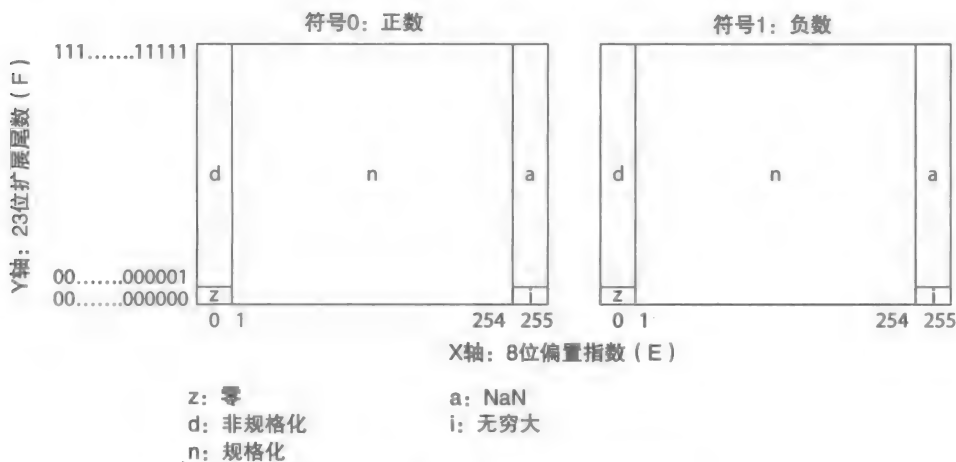


图 3-32 单浮点 FPN 的实例

这里 {} 用于表示级联

浮点加法算法：规格化数据空间

1) 将输入初始化: A 必须大于等于 B 。

i. 将 $A.F = \{1, A.F\}$ 和 $B.F = \{1, B.F\}$ 转换为内部（内存中）尾数表示。

ii. 保证 $|A| \geq |B|$ 。如果 $|A| < |B|$, 将 A 、 B 值交换; $|X|$ 表示 X 的绝对值。

2) 对齐小数点: 将 $B.F$ 右移 D 位。 D 的定义如下。

i. 让 $D = A.E - B.E$ 。

ii. 将 $B \cdot F$ 右移 D 位, 左边用 0 填充。

3) 生成结果 (R): 计算 $R.F = A.F \pm B.F$ 。

i. 生成 $R.F$, 作为 $A.F$ 与 $B.F$ 的和或差, 根据以下 $A.s$ 与 $B.s$ 的值判断进行的操作:

| A.s | B.s | 运 算 |
|-----|-----|-------------|
| 0 | 0 | $A.F + B.F$ |
| 0 | 1 | $A.F - B.F$ |
| 1 | 0 | $A.F - B.F$ |
| 1 | 1 | $A.F + B.F$ |

ii. 让 $R.s = A.s$ 且 $R.E = A.E$ 。

4) 将结果规格化: 如果 $R.F$ 未规格化, 将 $R.F$ 转换为 $1.F$ 的形式。

i. 如果 $R.F$ 的形式为 $1x.F$, 这里 x 在 3.i 步中不是 0 就是 1, 这样 $R.E$ 加 1 且 $R.F$ 右移一位, 得到 $1.F$ 形式; $1x.F$ 的 LSB (最低有效位) 将丢失。

ii. 或者, 如果 3.i 步中 $R.F$ 有前导零, 则 $R.F$ 每左移一位就将 $R.E$ 减 1, 为了消除前导零, 将 $R.F$ 转换为 $1.F$ 的形式。例如, 如果 $R.F = 0.01xxxxxx \cdots x$, 则将左移两位变为 $1.xxx \cdots x$ ($1.F$ 形式)。在这个例子中, $R.E$ 必须减 2。

5) 将结果四舍五入:

最终的结果, $S.F = \{1, S.F\}$, 较高位将从 $R.F$ 中选择相应的位数: 内部表示中单浮点数为 24 位, 双浮点数为 54 位。舍去 $R.F$ 未使用的较低位。然而, 舍去的位数可以用于四舍五入, 可能会导致在 $S.F$ 的最低有效位上加 1。如果四舍五入后的 $S.F$ 又变成了 $1x.F$, 那么需要重新将其规格化为 $1.F$ 的形式。对于四舍五入更完整的讨论还应包括在 IEEE 浮点标准中列出的 guard (G)、round (R) 和 sticky (S) 等位, 但已超出了本书涵盖的范围。

6) 最终输出:

在存入内存之前, $R.s$ 、 $R.E$ 和 $S.F$ 将连接成 32 位单浮点数或者 64 位双浮点数, 即

$$S = \{R.s, R.E, S.F\}$$

在上千万个运算中, $R.F$ 为 64 位, 且所有整数运算和位移函数都可以用于 64 位尾数中来将四舍五入错误减少到最小值。

例 3-9 $A = 17.875$ 且 $B = 15.75$, 计算 $S = A + B$ 。假设 A 、 B 和 S 都为具有 1 位符号位、

[133] 偏移量 = 63 的 7 位 e 指数和 8 位尾数的 16 位浮点数。

解: 计算 $S = A + B$ 的 5 个步骤如下:

1) 将实数 A 和 B 转换为二进制表示法。

2) 将二进制表示法转换为与其等价的非偏置指数科学形式。

3) 将非偏置指数科学形式转换为与其等价的偏置指数形式。

4) 将偏置指数形式转换为 16 位内存表示形式。

5) 遵循浮点加法算法, 将两个浮点数相加。

步骤 1: 转换为二进制

$$A = 10001.111 \times 2^0 \quad B = 1111.11 \times 2^0$$

步骤 2: 科学形式 (非偏置指数):

$$A = 1.0001111 \times 2^4 \quad B = 1.11111 \times 2^3$$

步骤 3: 科学形式 (偏置指数), IEEE 形式为 $1.F \times 2^E$

$$A = 1.0001111 \times 2^{4+63} \quad B = 1.11111 \times 2^{3+63}$$

$$A = 1.0001111 \times 2^{67} \quad B = 1.11111 \times 2^{66}$$

步骤 4: 内存中表示形式:

$A = 0x431E$ (0, 1000011, 00011110) $B = 0x42F8$ (0, 1000010, 11111000)

步骤 5：运用之前讨论的浮点加法算法：

1) 初始化输入：

因为 $|A| \geq |B|$ ，所以不需要将 A 和 B 互换。

$A.s = 0, A.E = 1000011$ (67), $A.F = 1.00011110$
 $B.s = 0, B.E = 1000010$ (66), $B.F = 1.11111000$

2) 对齐小数点：

$D = A.E - B.E$
 $= (1000011)_2 - (1000010)_2$ 或 $67 - 66 = 1$
 $= 0000001$

将 $B.F$ 右移一位 ($D = 1$) 得到 $B.F = 0.111111000$

3) 生成和：

$R.F = A.F + B.F$
 $= 1.00011100 + 0.111111000$
 $= 10.00011010$ (not in the 1.F format)
 $R.s = A.s = 0$
 $R.E = A.E = 67$

4) 规格化结果：

- i. $R.E$ 加 1 得到 $R.E = 68$ ($67 + 1$)。
- ii. $R.F$ 右移一位得到 $R.F = 1.000011010$ (为 1.F 形式)。

5) 结果四舍五入：

选择 $R.F$ 的高 9 位为 $S.F = 1.00001101$ 。 $R.F$ 的低位都为 0，所以忽略不计。结果为： 134
 $S.F = 1.00001101, S.F = 00001101$

(然而，如果 $R.F$ 为 1.000011011，其最低有效位为 1，则需要将 $S.F$ 四舍五入为 1.00001110，即 $1.00001101 + 0.00000001$ 。)

6) 最终结果：

$S = \{R.s, R.E, S.F\}$
 $S = \{0, 1000100, 00001101\}$ (或 $0x440D$ 在存储器中)

S 将转换为十进制数，如下所示：

$S = 1.00001101 \times 2^{68}$ (偏置科学表示)
 $S = 1.00001101 \times 2^{68-63}$ (转换成非偏置)
 $S = 1.00001101 \times 2^5$ (非偏置科学形式)
 $S = 100001.101$ (二进制)
 $S = 33.625$ (十进制)
 $= 17.875 + 15.75$

浮点减法、乘法和除法都是类似的过程。对于减法，尾数先对齐，与加法中一样，然后如果 $A.s = B.s$ 则相减， $A.s \neq B.s$ 则相加。对于乘法，尾数相乘，指数相加，且符号位进行异或运算。最后，对于除法，尾数进行整数除法，指数相减，符号位进行异或运算。四舍五入和规格化步骤与之前讨论的浮点数相加步骤相同。

然而，因为对于浮点数除法，被除数的内部尾数 $N.F$ 和除数的内部尾数 $D.F$ 最高有效位都为 1， $N.F$ 不用在左侧补零，如图 3-28 所示的整数除法 (参考练习 3.29)。

3.8.4 浮点单元

图 3-33 展示了浮点加法器的数据通路。数据通路包括实现算法中每一步骤的组合电路模型。在数据通路中, 如果 $|A| < |B|$ 则会用到交换模块。为了保证 $|A|$ 小于 $|B|$, $A.E$ 必须小于 $B.E$, 或者如果 $A.E = B.E$, $A.F$ 必须小于 $B.F$ 。交换模块(未展示)中的两个减法模块生成 $A.E - B.E$ 和 $A.F - B.F$ 的差值。这两个模块的借位信号表示 $|A| \geq |B|$ 是否为真。

如果 $|A| < |B|$, 则 A 和 B 必须进行交换, A 为较大的数, 为数据通路左边的输入。两个 2-1 的 MUX (未展示) 在需要时转换将 A 转换为 B 或者将 B 转换为 A 。如果 $A.E < B.E$ 或者当 $A.E = B.E$ 且 $A.F < B.F$ 时, A 和 B 两个输出需要进行转换。

在计算 $A.F \pm B.F$ 时, 右移模块用于调整小数点的位置。位移模块也用于每一个的规格化和四舍五入模块。组合位移将在后续讨论。

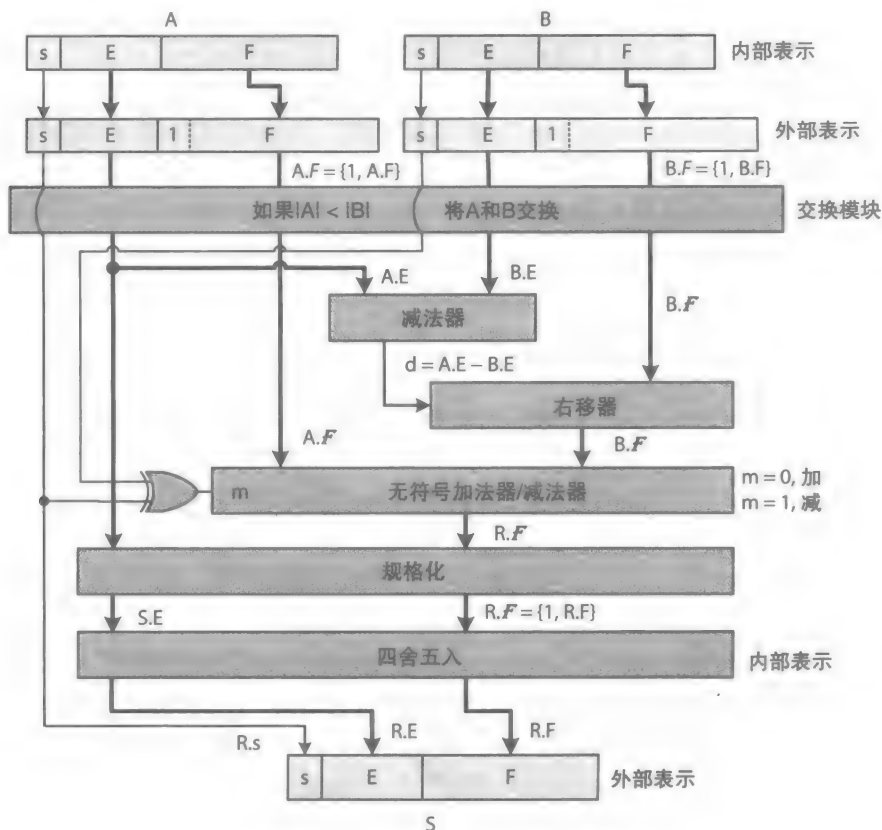


图 3-33 浮点加法器的数据通路

组合位移

组合位移器用 $\log_2(k)$ 个 2-1 MUX 实现, 组织成 $\log_2(k)$ 层次, 其中 k 表示数字需要移动的范围。例如, 对于 $k = 8$, 组合位移其可以将输入右移 $0 \sim k - 1$ 位或者 7 位。图 3-34 展示了 $k = 8$ 的 8 位组合右移器。3 位 $S = s_2s_1s_0$ 可以移动的范围是 $0 \sim 7$ 位。 s_2 、 s_1 和 s_0 为三个 MUX 的输入。

如图所示, 当 $s_0 = 0$ 时, 最高层 (第 0 层) MUX 选择 X 作为输出, 当 $s_0 = 1$ 时, 选择 X 右移一位作为输出。当 $s_1 = 0$ 时, 下一个 MUX 选择 Y 作为输出, 当 $s_1 = 1$ 时, 选择 Y 右移

S

两位作为输出。例如，当 $s_2 = 0$ 时，最后最底层（第 2 层）MUX 选择 Z 作为输出，当 $s_2 = 1$ 时，选择 Z 右移 4 位作为输出。

例如，当 $S = s_2 s_1 s_0 = (011)_2$ 时， $s_0 = 1$ 使得第 0 层的 MUX 将右移一位输出。信号 $s_1 = 1$ 使得第 1 层将 Y 右移两位作为输出。最后 $s_2 = 0$ 使得第 2 层 MUX 输出 Z 本身数值。作为结果，位移器将 X 右移了 3 位，当 $S = (101)_2$ 时， X 将右移 5 次。当 $S = (111)_2$ 时， X 可右移的最大次数为 7 次。

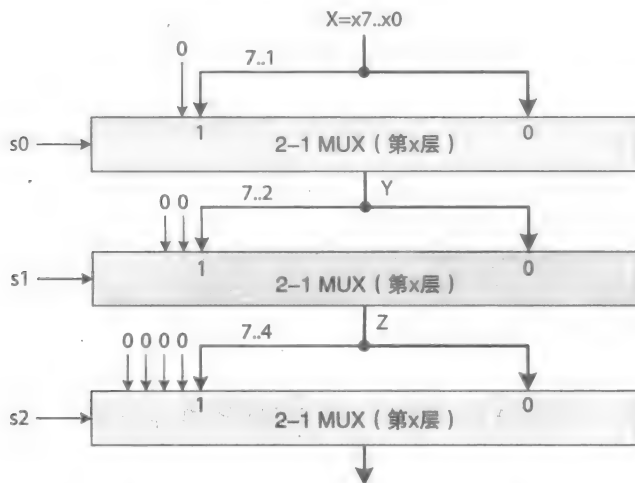


图 3-34 8 位填充零的组合右移器

136

参考文献

1. Vincent P. Heuring and Harry F. Jordan, *Computer Systems Design and Architecture*, Pearson Education, Inc., 2004.
2. Steve Wilson, "Alternative Subtraction Algorithms," http://www.sonoma.edu/users/w/wilsonst/courses/math_300/groupwork/altsub/default.html.
3. W. Kahan, "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic," <http://www.cs.berkeley.edu/~wkahan/ieee754status/>.
4. Nikrouz Faroughi, "A floating-point data space model: domain and range of a function," *WORLD COMP'03, The 2003 World Congress in Computer Science, Computer Engineering, and Applied Computing*, June 25-28, 2007, Las Vegas, USA.
5. Intel 64 and IA-32 Architecture Software Developer's Manual; Volume 1: Basic Architecture.

练习

3.1 现有一个 CPA (8)，8 位的 CPA，完成以下练习：

- a. 用非门和与非门设计一个加法器，并给出所需逻辑门的总数量。使用第 2 章给出的全加器 SOP 表达式实现。
- b. 给出所需三极管总数量。
- c. 设计一个 CPA (32) 需要多少三极管？

3.2 计算下列 2 的补码的和与差值。对于每个结果标明是否有溢出的情况。

- | | |
|-------------|-------------|
| a) | b) |
| A: 11001100 | A: 11111000 |
| B: 00110100 | B: 00001000 |

| | |
|-------------|-------------|
| + | + |
| c) | d) |
| A: 00000001 | A: 11111000 |
| B: 00000010 | B: 00001000 |
| - | - |
| e) | f) |
| A: 10000010 | A: 01111101 |
| B: 00000011 | B: 11111010 |
| - | - |

137

- 3.3 给出 1 位 PGU 和 1 位 CGU 的 POS 表达式并用非门和或非门画出它们的电路框图, 要求使用最少的逻辑门数量。
- 3.4 假设一个 8 位加法器使用两个 4 位 CPA (标记为 CPA1 和 CPA2) 组成的, 其中进位输出 c_3 作为 CPA2 的进位输入。CPA1 传入原始进位输入 c_{-1} 。为了提高加法器的运行速度, c_3 的生成公式为 $c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_{-1}$, 其中 p 和 g 可以在 0.3ns 内生成。给出新的加法器可以提升的速率值, $\Delta FAc = 0.5ns$ 。
- 3.5 设计和估计用以下模块实现的 16 位混合加法器的传输延迟:
- 2 个 CLA (8) 模块
 - 4 个 CLA (4) 模块
 - 8 个 CLA (2) 模块
- 3.6 通过计算 ΔCPA (8) 与 ΔCLA (8) 的比值比较 ΔCPA (8) 和 ΔCLA (8) 速率。使用全加器的 SOP 表达式。
- 3.7 用 4 位 BCGU 设计 15 位 CLA, 其中每一个 BCGU 输出 3 个进位以及 p^* 和 g^* 信号。
- 3.8 用 1 位 2-1 MUX 设计一个 1 位 8-1 MUX。假设与非门有 0.1ns 延迟, 给出其传输延迟的估计。
- 3.9 用 1 位 2-1 和 1 位 4-1 MUX 设计一个 1 位 8-1 MUX。假设与非门有 0.1ns 延迟, 4-1 MUX 用 SOP 表达式实现, 同时估计其传输延迟。
- 3.10 估计图 3-16 中 8 位 ALU 的传输延迟, 假设与非门有 0.1ns 延迟, 加法器用 CLA (8) 实现, 且只有 2-1 MUX 可用。
- 3.11 证明图 3-22 中的电路可以实现 ALU 映射模块。
- 3.12 现有一个 8 位并行 ALU, 其只能实现 4 个功能: 加法、减法、按位与和异或。假设一个加法器 / 减法器模块用一个由 CLA (2) 模块组成的混合加法器设计。此外, 当进行按位计算时溢出标记必须屏蔽。假设只有 8 位 2-1 MUX 可以选择。完成以下练习:
- 画出数据通路并估计其传输延迟, 假设非门和与非门有 0.1ns 延迟。
 - 给出其映射模块的真值表。
- 3.13 用 2-1 译码器模块设计一个 4-2 译码器。提示: 你也许还需要一个 1 位 2-1 MUX 和一个 2 输入或门。
- 3.14 用 4-2 译码器模块设计一个 8-3 译码器。提示: 你也许还需要一个 2 位 2-1 MUX 和一个 2 输入或门。
- 3.15 用表 3-8 中给出的 1 位 ALU 片真值表设计图 3-23 所示的 8 位串行 ALU。
- 3.16 设计一个 2 位 2 的补码比较器, 且用 4 个比较器模块来设计 8 位比较器。提示: 一个 2 位 2 的补码比较器输入两个 2 位输入 A 、 B 和 gti (大于)、 eqi (等于) 和 lti (小于) 信号, “ i ” 表示从之前的模块输入, 输出信号为 gto 、 eqo 和 lto , “ o ” 表示输出。首先设计一个只有 A 和 B 的 2 位 2 的补码比较器生成三个输出, 当 $A > B$ 时输出为 gtt 、 $A = B$ 时输出为 eqt 、 $A < B$ 时输出为 ltt , 这里 “ t ” 表示最初的。然后将 gtt 、 eqt 、 ltt 与 gti 、 eqi 、 lti 组合起来生成 gto 、 eqo 和 lto 。例如,

138

当 ($gti = 1$ 且 $gtt = 1$) 或 ($gti = 1$ 且 $eqt = 1$) 或 ($eqi = 1$ 且 $gtt = 1$) 或 ($lti = 1$ 且 $gtt = 1$) 时, $gto = 1$ 。

3.17 现有图 3-26 给出的 4 位阵列乘法器。完成以下练习：

- a. 根据进位与和值的传输延迟估计其传输延迟；即根据 ΔFAc 和 ΔFAs 。
- b. 给出 n 位阵列乘法器的传输延迟公式。

3.18 现有 8 位阵列乘法器，CLA (8) 替换了最后一行的 CPA (8)。假设 $\Delta FAc = 0.2ns$ ， $\Delta FAs = 0.3ns$ 且 $CLA(8) = 0.8ns$ ，问新的乘法器可以比原来的乘法器运行多快？

3.19 用恢复除法算法运算用 $N = 10101101$ 除以 $D = 1110$ 。你可以使用计算器进行二进制运算来验证你的结果。

3.20 首先通过设计一个 1 位组合减法 MUX 位片来设计一个阵列除法器，使得传输延迟最小（例如，SOP 或者 POS 表达式）；然后用其来设计 4 位位串行减法 MUX 模块；且用其替换图 3-28 中的 4 位减法和 4 位 MUX 对。

3.21 用你熟悉的语言（或者用 Excel）写出实现以下倒数除法算法的程序，且当 $D = 1.0$ 时做一次观察。然后证明倒数除法算法，计算 Q ，等于 N/D ，用除法运算 ($/$) 计算。倒数除法算法如下：

```
#define ITERATIONS 15
Procedure ()
{
    float D, N, Q;
    float R;
    input D, N; //e.g., D = 1.99 and N = 2.4
    For j = 1 to ITERATIONS do
        R = 2 - D;
        D = D * R;
        N = N * R;
        print j, D, N, R
    Endfor
}
```

考虑到在计算 Q 的过程中没有除法操作；在 N 除以 D 中只用到了乘法操作和减法操作。这个算法是在硬件 Intel x486 处理器中作为浮点除法指令存在的。执行两次程序，一次用输入 $D = 1.99$ 且 $N = 2.4$ ，一次使用输入 $D = 1.56$ 且 $N = 2.4$ 。考虑到 D 的尾数总是小于 2（例如， D 的最大尾数的二进制表示为 $1.1111111\cdots 1 < 2$ ，“浮点”类型中小数点后有 23 个 1，“双浮点”类型中小数点后有 52 个 1）。

139

当 D 变为 1.0 时比较 N 值和用原始 $N(N_0)$ 除以 ($/$) 原始 $D(D_0)$ （例如 $Q = 2.4/1.99$ ）求得的 Q 值。对于某些 i 当 D_i 变为 1.0 时，比较 N_i 和 $Q = N/D$ 。对于某些 i ，当 $D_i = 1.0$ 时可以得到 $Q = N_i$ 吗？这两次运行你注意到什么？

3.22 给出 6.725 的 IEEE 浮点表示。

3.23 给出 0.35 的 IEEE 单浮点类型表示。

3.24 $0x41DD0000$ 是 IEEE 单浮点数。在十进制中其表示的数值为多少？

3.25 给出两个单浮点数 $A = 0xC18D0000 = -17.625$ 和 $B = 0x41080000 = 8.5$ 相加求出和值 (S) 的步骤。

3.26 设计一个 8 位组合右移运算器。每一次数值进行一次运算右移，符号位不变。且给出 -80 右移三次的步骤 ($-80 \ggg 3$)。

3.27 用 CPA (8) 设计一个 8 位 2 的补码的加法器 / 减法器 Verilog 行为和结构模块并进行仿真。用

“case”语句来描述一个全加器；然后用一个 Verilog 结构模块来设计 CPA (8)，使用 8 个全加器副本。用 “assign” 语句来溢出标记 (ovf) 的表达式，用 “if-else” 语句来描述转换模块。设计一个 Verilog 测试模块来测试你的设计是否正确，使用测试向量 $0x80 - 0x01$ 和 $0x7F + 0x01$ 。在两个例子中溢出标记是否为真？

- 3.28 用 Verilog 设计一个 8 位恢复除法器。用一个 Verilog 行为模块设计一个 1 位与全加器相似的减法器。且设计一个 8 位与 8 位 CPA 类似的 BPS。设计一个 8 位 2-1 MUX 的行为描述。将若干个 BPS 和 MUX 模块结合起来用于设计除法器。设计一个测试模块来测试你的设计。
- 3.29 考虑外部尾数 $N.F = \{1, N.F\}$ 的浮点被除数和内部尾数 $D.F = \{1, D.F\}$ 的浮点除数，其中 $N.F$ 和 $D.F$ 都为 4 位数值。用练习 3.20 中 4 位位串行减法 MUX 设计一个 FPU 中的整数除法器。提示： $N.F$ 将从右边填充 0；这个例子中，在每一步中 A_k 总是一个 5 位数值和 $A_k[4]$ ，最高有效位 (MSB)，不用于确定下一位数 (例如 $A_k[3:0] - D.F$)；在每一步中 $A_k[4]$ 用于与借位输出 (bo) 连接来决定下一个商位 $q_k = A_k[4] + (\overline{bo_k})$ 。

计算机安全

- 3.30 计算机安全 (硬件木马)：练习 11.12 了解计算机恶意电路如何设计。不用实现其触发机制；直接操作 MUX 来引起计算机攻击 (参照 11.2 节)。
- 3.31 计算机安全 (访问控制)：练习 11.26 设计一个适用于硬件实现的分级访问控制模块 (参照 11.1.4 节和 11.1.5 节)。

时序电路：核心模块

4.1 简介

尽管组合电路是必要的而且也是数字系统中重要的一部分，但它的输出仅仅取决于当前的输入。希望输入中的任何变化都会改变当前的输出。而在另一方面，时序电路的输出不仅取决于当前的输入，还受之前输入的影响。例如，如图 4-1 中的描述，如果使用一个单端加法器去计算出几个数的和，那么必须要有一个初始值为 0 的部分和被存储起来，而且能够和一个新的数值相加产生下一个部分和。因此，这个电路是个时序电路，因为当前的部分和是由之前输入的所有数值进行相加得到的，而且，下一个部分和的值取决于被加的下一个数值，即现在被输入到加法器的值，也是当前被送到部分和的值。

通常情况下，一个复杂的时序的数据通路要求要有组合电路去产生结果输出，还要有一个存储模块（通常是寄存器）来保存这些结果。数据通路还需要一个遵循一组特定步骤（即算法）的控制单元（控制器）以便通过数据通路实现复杂的功能。控制单元是一个时序电路，它使用存储模块来保存它当前的状态（如算法中的具体步骤），这样就能够根据现在接收到的输入去决定它的下一个状态。

时序电路还要求有一个时序控制信号，称为时钟。它用于去确定在什么时间一个值（比如图 4-1 中加法器的和）应当被保存在存储模块中。注意，加法器不会在同一时刻产生和的每一位。和的每一位都是由一个组合电路在一个特定的传播延时后输出的。传播延时小的电路输出的结果将会快于传播延时大的电路。因此，加法器在经过它内部最大的传播时延之后产生的输出和才是有效的。否则，和的一位或多位仍在变化中，因此和是无效的。时钟信号给出和的有效时间，然后让存储模块把它保存下来。

141

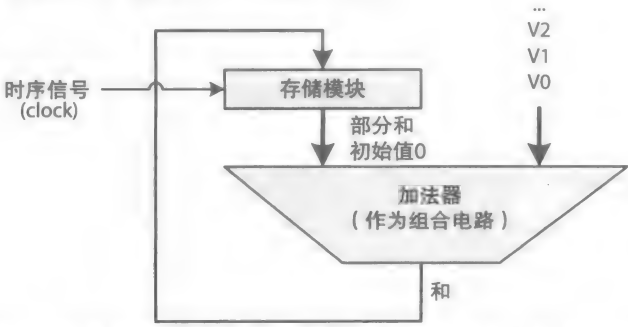


图 4-1 时序电路示意图；它计算出几个数值的和，如 V_0 、 V_1 等

这一章节涉及了设计存储模块所需要的核心逻辑电路。在核心电路中，输出反馈回来又作为输入，因此该电路可以保留住一个输出值，于是就产生了一个存储模块。但是，这些核心电路要求有严格的操作约束，以使输出稳定并保留住存储的值。它们在小型和大的时序电路中会被用到。小的时序电路设计在第 5 章中进行介绍，大的时序电路设计在第 6 章介绍，CPU 设计在第 8 章中介绍。

4.2 SR 锁存器

图 4-2 描述了 SR 锁存器的电路组成。这个电路有两个输入端： s 和 r 。它们分别作用于置位 q ，把 q 置为 1，或复位 q ，把 q 置为 0。信号 p 和 q 是相互作用的，作用关系分别表示为 $q = r + \overline{p}$ ， $p = \overline{r} + q$ 。想要确定 q 的值，就必须知道 p 的值，反之亦然。因此，由于 p 和 q 的初值不确定，电路的最终结果应当对以下 4 种情况进行分析得知。

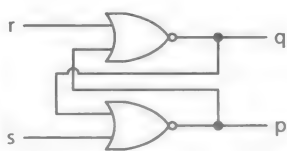


图 4-2 基本 SR 锁存器

情况 1: $s = 0, r = 0$

a. 假设当前的 $q = 0$ ，让 $s = 0, r = 0$ ，确定 p 和 q 的值。在这种情况下，

$$\begin{aligned} p^{\text{new}} &= \overline{s + q} \\ &= \overline{0 + 0} \\ &= 1 \\ q^{\text{new}} &= \overline{r + p^{\text{new}}} \\ &= \overline{0 + 1} \\ &= 0 \end{aligned}$$

b. 假设初始值 $q = 1$ ，让 $s = 0, r = 0$ ，确定 p 和 q 的值。

$$\begin{aligned} p^{\text{new}} &= \overline{s + q} \\ &= \overline{0 + 1} \\ &= 0 \\ q^{\text{new}} &= \overline{r + p^{\text{new}}} \\ &= \overline{0 + 0} \\ &= 1 \end{aligned}$$

因此，当 $s = 0, r = 0$ 时， $q^{\text{new}} = 0$ （未改变）， $p^{\text{new}} = 1$ 。

因此，当 $s = 0, r = 0$ 时， $q^{\text{new}} = 1$ （又未改变）， $p^{\text{new}} = 0$ 。

第 1 种情况表明，当 $s = 0$ 而且 $r = 0$ 时， p 和 q 的值保持不变，而且 $p = \overline{q}$ 。

情况 2: $s = 0, r = 1$

a. 假设 $q = 0$ ，让 $s = 0, r = 1$ 。此时，

$$\begin{aligned} p^{\text{new}} &= \overline{s + q} = \overline{0 + 0} = 1 \\ q^{\text{new}} &= \overline{r + p^{\text{new}}} = \overline{1 + 1} = 0 \end{aligned}$$

因此，当 $s = 0, r = 1$ 时， $q^{\text{new}} = 0$ （未改变）， $p^{\text{new}} = 1$ 。

b. 假设 $q = 1$ ，让 $s = 0, r = 1$ 。

$$\begin{aligned} p^{\text{new}} &= \overline{s + q} = \overline{0 + 1} = 0 \\ q^{\text{new}} &= \overline{r + p^{\text{new}}} = \overline{1 + 0} = 0, \quad q \text{ 发生变化} \\ p^{\text{new}} &= \overline{s + q^{\text{new}}} = \overline{0 + 0} = 1, \quad p \text{ 发生变化} \\ q^{\text{new}} &= \overline{r + p^{\text{new}}} = \overline{1 + 1} = 0 \quad q^{\text{new}} \text{ 未发生变化, } p \text{ 和 } q \text{ 保持不变} \end{aligned}$$

在这种情况下， q 的值发生变化，但最终保持稳定，而且保持为 0。

第 2 种情况表明，无论 q 的初值是多少，当 $s = 0, r = 1$ 时， $q^{\text{new}} = 0, p^{\text{new}} = 1$ 。因此， q 被复位为 0（或者说这个锁存器锁存逻辑 0），而且 $p = \overline{q}$ 。

情况 3: $s = 1, r = 0$

a. 假设 $q = 0$ ，让 $s = 1, r = 0$ 。

$$p^{\text{new}} = \overline{s + q} = \overline{1 + 0} = 0$$

142

143

$$q^{\text{new}} = \overline{r + p^{\text{new}}} = \overline{1 + 0} = 0 \quad q \text{ 发生变化}$$

$$p^{\text{new}} = \overline{s + p^{\text{new}}} = \overline{1 + 1} = 0 \quad p \text{ 和 } q \text{ 的值已经稳定}$$

因此，如果 $s = 1, r = 0$ ，则 $q^{\text{new}} = 1$ (从 0 变为 1)， $p^{\text{new}} = 0$ 。

b. 假设 $q = 1$ ，让 $s = 1, r = 0$ 。

$$p^{\text{new}} = \overline{s + q} = \overline{1 + 1} = 0$$

$$q^{\text{new}} = \overline{r + p^{\text{new}}} = \overline{0 + 0} = 1 \quad \text{保持不变}$$

第 3 种情况表明不论 q 的值是多少，当 $s = 1, r = 0$ 时， $q^{\text{new}} = 1, p^{\text{new}} = 0$ 。因此， q 被置位为 1 (或者说锁存器锁存逻辑 1)，而且 $p = \bar{q}$ 。

情况 4: $s = 1, r = 1$

a. 假设 $q = 0$ ，让 $s = 1, r = 1$ 。

$$p^{\text{new}} = \overline{s + q} = \overline{1 + 0} = 0$$

$$q^{\text{new}} = \overline{r + p^{\text{new}}} = \overline{1 + 0} = 0$$

这种情况下， q^{new} 和 p^{new} 都变为 0。

b. 假设 $q = 1$ ，让 $s = 1, r = 1$ 。

$$p^{\text{new}} = \overline{s + q} = \overline{1 + 1} = 0$$

$$q^{\text{new}} = \overline{r + p^{\text{new}}} = \overline{1 + 0} = 0 \quad q \text{ 发生变化}$$

$$p^{\text{new}} = \overline{s + p^{\text{new}}} = \overline{1 + 0} = 0 \quad p \text{ 和 } q \text{ 现在保持稳定}$$

在这种情况下， q^{new} 和 p^{new} 又一次都变为 0。

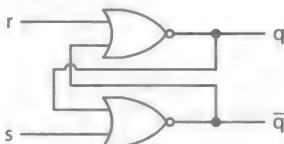
第 4 种情况比较特殊，它表明无论 q 的初值如何，当 $s = 1, r = 1$ 时， q^{new} 和 p^{new} 都变为 0，与其他 3 种情况有所不同。与前面 3 种情况对比，这种情况使得 p 和 q 产生了矛盾的结果，不再使得 p 和 q 的结果保持相反。这种情况的矛盾还体现在：如果使 r 和 s 同时由 1 变为 0，信号 p 和 q 将会振荡，将同时变为 1，然后再同时变为 0，然后一直这样重复下去。尽管在仿真中这种振荡会一直持续下去 (永远不会稳定)，但在实际电路中这种振荡将最终停下来。而这个最终的稳定结果将会是随机的，0 或 1，而且满足 $p = \bar{q}$ 。第 4 种情况可以改变系统的状态，随机地造成影响。因为这个原因，第 4 种情况不应当出现。也就是说， r 和 s 永远都不要同时变为 1。

情况 1 ~ 3，从另一方面讲，正好提供了一个存储模块所需的必要功能：保持 q (第 1 种情况)，复位 q 或存储 0 (第 2 种情况)，置位 q 或存储 1 (第 3 种情况)，而且 p 始终等于 \bar{q} 。

图 4-3 说明了 SR 锁存器的最终电路，其中 p 由 \bar{q} 代替。它的状态表 (也叫真值表) 也在图中表示了出来，而第 4 种情况 ($s = 1, r = 1$) 被标记为未使用。在这个表中， q 的初始值被表示为 q' ，意思是说 q 在 t 时刻的值。 q 的新 (下一个) 值被表示为 q^{t+1} ，意味着 q 和 \bar{q} 的稳定值。

| s | r | q^{t+1} |
|---|---|-----------|
| 0 | 0 | q' |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | — |

a)



b)

图 4-3 SR 锁存器和它的状态表：a) SR 锁存器的状态表；b) 或非门 SR 锁存器

这个 SR 锁存器电路在成为一个真正有用的电路之前还缺少一些特点：

- 我们必须保证信号 r 和 s 不会同时置高（换言之，第4种情况不会发生）。
- 在系统启动期间，我们必须可以初始化 q 为一个可知的值，0 或 1。

时钟触发 SR 锁存器

时钟是一个被称为振荡器的电子设备，这个振荡器以一定的周期不断重复地输出 1 和 0。它被用于在特定的时间进行采样，而且当时钟变为 0 或变为 1 时采样结束。图 4-4 代表了一个时钟触发的 SR 锁存器的电路。这个时钟信号分别和 r 、 s 信号进行了与操作。由它控制在何时 r 和 s 信号被允许去改变这个核心电路的状态（图 4-3b），即 q 和 \bar{q} 的值。

当图 4-4 中信号 $\text{clk} = 0$ 时，电路中 G1 和 G2 两个与门将会输出 0，与 r 和 s 的值没有关系。这样让核心 SR 锁存器继续保持它的当前值（第 1 种情况）。然而，当 clk 变为 1 时，G1 会输出 r 的当前值，G2 也会输出 s 的当前值。这时，根据图 4-3a 中的状态表可知， r 和 s 的值将会改变核心 SR 锁存器的状态。

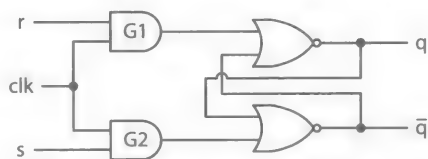


图 4-4 高电平 SR 锁存器

图 4-4 的电路中，如果当 $\text{clk} = 1$ 时，信号 r 和 s 的值才可以“进入”这个核心电路，则这个电路叫作高电平 SR 锁存器。同理，当 $\text{clk} = 0$ 时信号才能“进入”电路，这个电路叫作低电平 SR 锁存器。图 4-5 描述了一个有一点时延的高电平 SR 锁存器的场景，在这个场景中假设信号 s 比 r 的传播时延更小一些。

在时间段 1 中，信号 s 和 r 都在变化中，由于 s 的传播时延比 r 更小一些，在 r 仍然是 1 的时候， s 开始变为 1。因此如图中所示，在时序图的阴影部分， r 和 s 有一个短暂的都为 1 的状态（第 4 种情况）。然而，由于这个短暂的时间内的 $\text{clk} = 0$ ，G1 和 G2 两个与门都会输出 0，阻止了 $s = 1$ 和 $r = 1$ 进入这个核心电路，因此，这个锁存器保持了它的当前状态，没有受到影响。

在时间段 2 中，当 $\text{clk} = 1$ ， $r = 0$ ， $s = 1$ 时，这两个与门会把 $r = 0$ 和 $s = 1$ 送入这个核心电路中，使 q 变为 1。在这个时间段中，我们称这个锁存器对它的输入信号 r 和 s 进行采样。

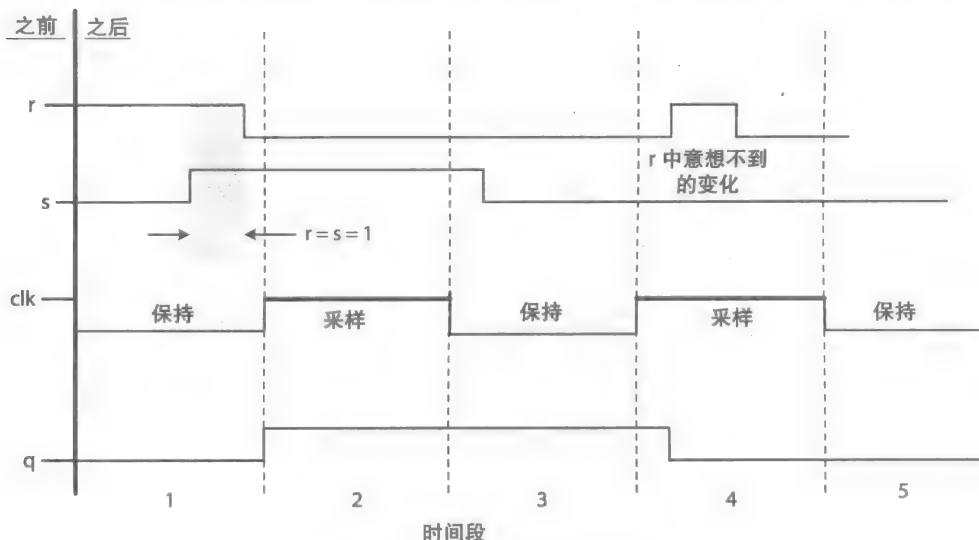


图 4-5 SR 锁存器时序举例

在时间段 3 中，当 $\text{clk} = 0$ （保持）时，锁存器保持 $q = 1$ ，结果保持为 1。最后，在时间段 4 中，当 $\text{clk} = 1$ 而且锁存器正在采样时， r 值一个瞬时的变化（例如电子脉冲）使 q 值不可预料地变为了 0。由于这个原因， r 和 s 的值在每一个采样期间都必须保持稳定，这是 SR 锁存器的一个缺点。

SR 锁存器的另外一个缺点是它需要两个输入端（ r 和 s ）进行控制。在现代集成芯片（ICs）的今天尤其如此，因为线路也会占用芯片的空间。现代的集成芯片使用成千上万的锁存器来生成寄存器。如果每一个锁存器都需要两个输入端去控制，这将占用两倍的空间去连接这个端口。

由 SR 锁存器设计实现的 D 锁存器解决了这些缺点。D 锁存器将在下一节中进行讨论。

由于输入信号 r 和 s 仅在采样时间段内能够对核心电路产生影响，它们分别被称为同步复位和同步置位信号。然而，独立于时钟信号，异步地初始化锁存器为 $q = 0$ 或 $q = 1$ 通常是有必要的。图 4-6 表示了一个带有异步重置和异步预置信号的时钟触发 SR 锁存器。这些信号是直接接到或非门的输入信号，而且，当每一个有效时，都可以独立地改变 q 和 \bar{q} 的值。当 $\text{reset} = 1$ ， $\text{preset} = 0$ 时， q 变为 0，与 r 、 s 和 clk 信号的状态无关。同样，当 $\text{preset} = 1$ ， $\text{reset} = 0$ 时， q 变为 1。 reset 和 preset 信号不能同时有效；只能有一个有效的信号将 q 的值初始化为 0 或 1。

与非类型

图 4-6 中或非类型的 SR 锁存器很容易理解，因为它使用了高电平有效信号。一个等效的与非类型的锁存器在图 4-7 中描述了出来，并且带有异步的低电平有效的重置（ _reset 或 _r ）信号和预置（ _preset 或 _s ）信号。注意，与非类型相比，与非类型锁存器中 r 和 s 的意义有所不同；现在 s 信号是和 q 信号保持一致， r 信号和 \bar{q} 信号保持一致。这个图同样是一个时钟触发 SR 锁存器的代表，其中输入端 c 代表了“时钟”。除非另有说明，否则“SR 锁存器”这个术语仅代表了时钟触发 SR 锁存器，而不论低电平还是高电平有效。

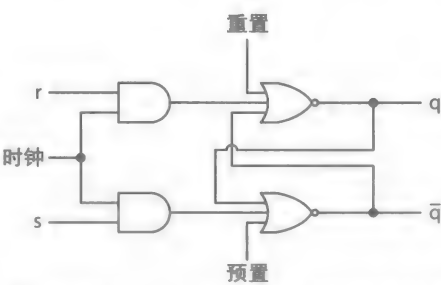


图 4-6 带有异步重置和预置信号的 SR 锁存器

147

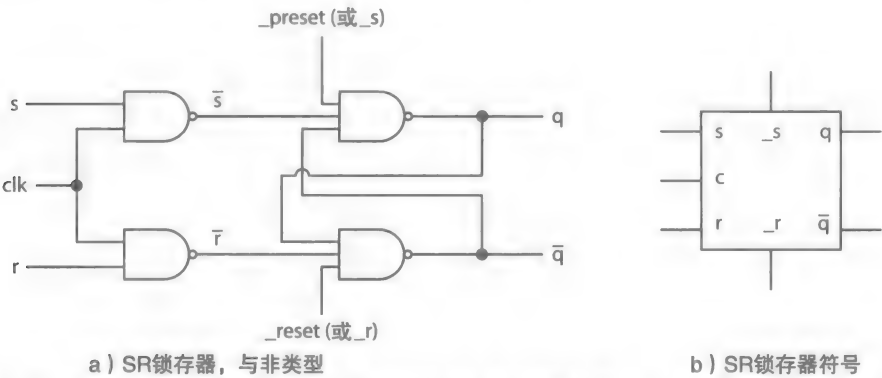


图 4-7 高电平 SR 锁存器：a) 仅有与非门的 SR 锁存器；b) SR 锁存器逻辑符号

4.3 D 锁存器

SR 锁存器通过消除保持选项（第 1 种情况）可以转换为 D 锁存器，这种选项中的 s 和

r 信号都是 0。在同步模式中,除了时钟信号, D 锁存器仅需要一个输入信号 d 来控制。这个可以通过把 d 连到 s 上,把 \bar{d} 连到 r 上来实现,如图 4-8 所示; d 代表了数据。D 锁存器仅工作在两种模式中,当 $d=1$ 时同步地置位 q ($q=1$) 或者当 $d=0$ 时同步地复位 q ($q=0$)。当时钟信号处于采样时期,信号 q 始终等于 d 。然而,就像 SR 锁存器一样,当时钟信号处于“关”状态不进行采样时, D 锁存器保持 q 的值。

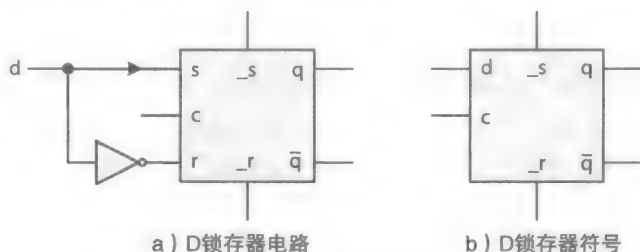


图 4-8 D 锁存器的电路和逻辑符号

4.4 锁存器的缺陷

无论 SR 锁存器还是 D 锁存器都有一种缺陷,在大多数的时序电路设计中限制了它们的应用。当两个或多个锁存器在同一个时钟信号下工作时,它们的 d 信号中没有可以依赖的数据存在。试想,在同一个时钟信号工作下的两个或多个锁存器,在这种情况下,这些锁存器中的 d 输入信号没有一个可以成为任何 q 或 \bar{q} 的关系值。否则,一种称为信号延伸的情况将会导致电路不能正常工作。

148

例如,如图 4-9 所示两个高电平 D 锁存器。这两个锁存器都在同一个时钟信号 clk 下工作。这种情况下, d_0 的值取决于 q_1 (即 $d_0 = q_1$)。这个电路本被作为一个两位右移的寄存器使用,但它不能工作。每次当 clk 由 0 变为 1 时,由信号 q_1 和 q_0 所决定的寄存器的初值会变为它的新值,即 $q_1^{new} = 0$, $q_0^{new} = q_1^{current}$ 。

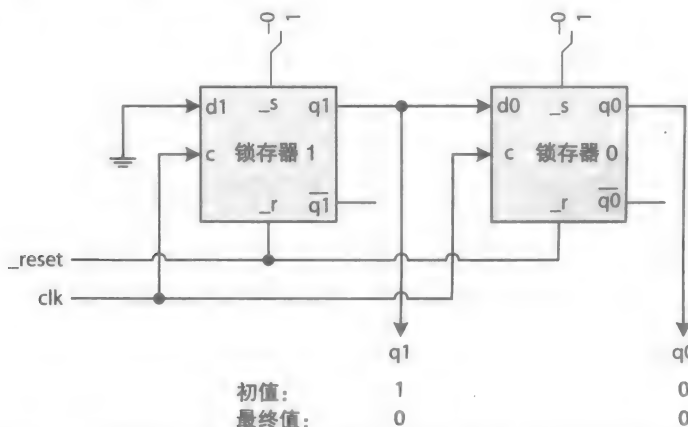


图 4-9 一种错误性设计的两位右移寄存器的图解;没有可用的锁存器

为了解释这些,假设通过使用低电平有效的 $_reset$ 信号复位锁存器,将它们的 q 值变为 0 之后,再通过连接到锁存器 1 上的低电平有效异步置位输入端 ($_s$) 将 q_1 置位为 1。这样就实现了相连的值 q_1 和 q_0 ,或者说 $q_1q_0 = (10)_2$,在图中所示作为 q_1 和 q_0 的初值。现在当

clk 由 0 变为 1 时， q_1q_0 的下一个值应当是 $(01)_2$ ，意味着 $d_1 = 0$ 从左端被右移到内部。然而锁存器将不会实现这些。

具体地说，就是当 clk 变为 1，两个 D 锁存器将同时对它们的输入端进行采样，将它们的当前值 $q_1^{\text{current}} = 1$ 和 $q_0^{\text{current}} = 0$ 改变为 $q_1^{\text{new}} = d_1 = 0$ 和 $q_0^{\text{new}} = d_0 = q_1^{\text{current}} = 1$ 。但是如果 clk 保持为 1，D 锁存器会继续对它们的输入端进行采样，使得 q_1^{new} 赋值给 q_0^{new} ，导致最终的结果 $q_1q_0 = (00)_2$ ，而不是预想的 $(01)_2$ 。在这种情况下，如果 clk 长时间保持为 1，信号 $d_1 = 0$ 延伸，导致 d_0 变为 0。

通常而言，通过使用需要依赖性输入的锁存器进行内部相连来实现独立性的控制是不可能的，因此，在很多重要的时序电路设计中是不能使用的，比如移位寄存器和控制单元。能够防止这种信号延伸的电路叫作触发器。

4.5 D 触发器

如图 4-10 中 D 触发器所示，触发器可以由两个相连的锁存器设计而成。在每一个时钟电平中，只有一个锁存器对它的输入信号进行采样，另一个锁存器保持它的当前值。这两个锁存器工作起来就像一个双门入口系统，就像很多建筑中用到的那样。任何时刻都只有一个门是打开的，另外一扇门处于关闭状态。一个人要进入这个建筑之中，必须先穿过第一扇门，再穿过第二扇门。在这个图中，这两个锁存器分别被标记为 A (A 门) 和 B (B 门)。它们的输入输出分别被记为 A.d、A.q、B.s 等。A 和 B 都是高电平有效锁存器，但工作在互补的时钟电平下。A 锁存器是一个 D 锁存器，B 锁存器是一个 SR 锁存器。

149

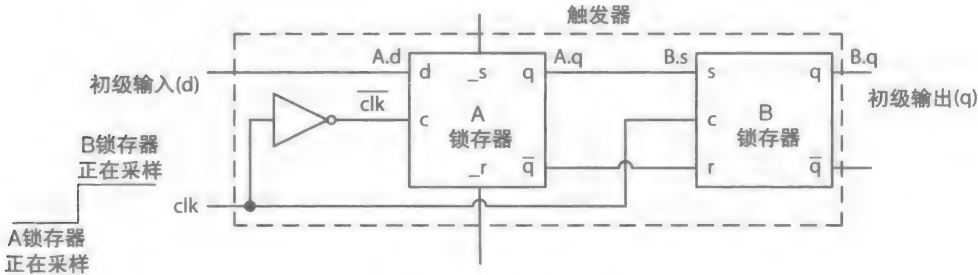


图 4-10 D 触发器

当 $\text{clk} = 1$ ， $\overline{\text{clk}} = 0$ 时，锁存器 A 不会进行采样（也就是 A 门关闭），保持它的 q 值，A.q。这段时间里，初级输入 d ($A.d = d$) 的变化不会改变 A.q 和 A.q 的值。这两个信号分别连接到了 B 锁存器的输入端 B.s 和 B.r，因此，如果 d 发生变化，即使 $\text{clk} = 1$ 时，B 锁存器正在采样（也就是 B 门打开），B.q 也不会发生变化。

然而，当 clk 由 1 变为 0， $\overline{\text{clk}}$ 由 0 变为 1 时，这两个锁存器交换模式；锁存器 A 开始采样 A.d（也就是 A 门打开），锁存器 B 停止采样 A.q（也就是 B 门关闭）；因此，锁存器 B 保持 B.q 值。在这段时间里，初级输入 d 的任何变化将会改变 A.q，但不会改变 B.q。因此，这两个锁存器工作在不同的时钟电平下，阻止了信号延伸。只有在 clk 再次由 0 变为 1 时，锁存器 A 的输出才可以影响锁存器 B 的输出。所以，触发器需要一个时钟边沿进行控制。

在这个图中，当 clk 信号由 1 变为 0 再变为 1（一个 1-0-1 的变化）时，触发器进行采样，然后保存它的初级输入 d 作为它的初级输出 B.q。

4.5.1 选择电路

图 4-11 表示了一个不同的总门数更少的 D 触发器。这个 D 触发器工作起来就像图 4-10 中所示的那样。当 clk 进行 1-0-1 变化的时候, 它对初级输入 d 进行采样。特别地, 当 clk 由 1 变为 0 时, 信号 \bar{s} 和 \bar{r} 变为 1, 触发器保持 q 值, 不受当前 d 值的影响。当 clk 由 0 变为 1 时, 触发器开始对 d 进行采样。如果 $d = 1$, 那么 $\bar{s} = 0$, $\bar{r} = 1$, q 将变为 1; 否则, 如果 $d = 0$, 那么 $\bar{s} = 1$, $\bar{r} = 0$, q 将变为 0。 \bar{s} 和 \bar{r} 的值在 clk 的下一个 1-0-1 变化到来之前保持不变。

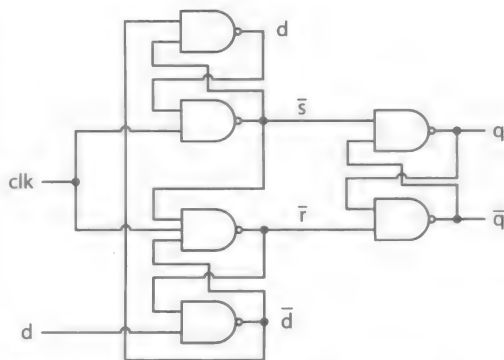


图 4-11 带有更少门电路的选择 D 触发器

4.5.2 操作规范

[150]

图 4-10 中锁存器 A 和 B 工作 (即采样) 在不同的时钟电平下 (0 或 1)。当一个锁存器正在采样 (门开启) 时, 另一个锁存器始终处于不采样的状态 (门关闭)。只有当 clk 出现 0-1 变化时, 被采样得到的输入信号 d 才会通过锁存器 A 到达锁存器 B (即 $B.q$), 此时是正在采样的锁存器 A 要停止采样, 锁存器 B 开始采样 $A.q$ 。注意现在的锁存器 A 已停止采样, 即使 d 发生变化, $A.q$ 也不会再改变。锁存器 A 下一次开始采样是在 clk 出现 1-0 变化的时候, 这时锁存器 B 停止采样。

D 触发器中, 数据从一个锁存器传输到另一个是在时钟边沿发生的, 而不是在时钟电平上; 因此, D 触发器被称为边沿触发。如果时钟 1-0-1 的变化可以让触发器对 d 进行采样 (在 1-0 变化时 $A.q = d$) 并保存为 q (在 0-1 变化时 $B.q = A.q$, 正触发, 箭头向上), 那么它就叫作正触发器或上升沿触发器。否则, 如果时钟 0-1-0 的变化可以让触发器对 d 进行采样 (在 0-1 变化时 $A.q = d$) 并保存为 q (在 1-0 变化时 $B.q = A.q$, 负触发, 箭头向下), 那么它就叫作负触发器或下降沿触发器。如图 4-12 所示, 边沿触发器的时钟输入端被一个右箭头符号作了标记 (>), 而且, 如果是下降沿触发器, 还会有一个小圆圈。

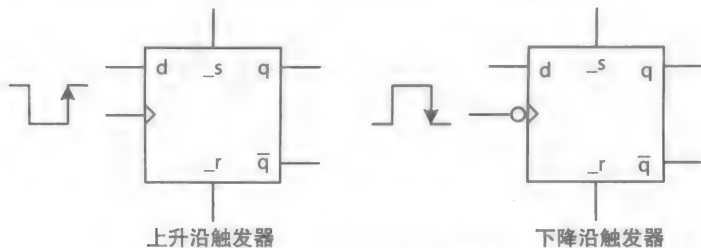


图 4-12 上升沿和下降沿触发器符号

4.5.3 建立和保持时间

与诸如 SR 触发器之类的器件不同, D 触发器是由高电平和低电平的锁存器设计而成 (但不被今天所用), 它有一个优点, 就是当锁存器 A (第一个锁存器) 仍在采样时, 允许它的 d 输入端发生变化。唯一的要求就是 d 信号能够趋于稳定, 而且在时钟发生变化时能保持短暂

的稳定。

例如，观察图 4-10 中上升沿触发的触发器。当 $\text{clk} = 0$ 时，锁存器 B 保持 $B.q$ 不变（也就是 B 门关闭），而且因为 $\overline{\text{clk}}$ 是 1，锁存器 A 采样 $A.d$ （即 A 门打开）。此时，当 clk 出现上升沿（0-1）变化成为 1 时， clk 和 $\overline{\text{clk}}$ 都保持为 1，直到 $\overline{\text{clk}}$ 经过非门短暂的传播延时变为 0，并使锁存器 A 停止对 d 进行采样（即 A 门关闭）。在这个短暂的时间里，每个锁存器都分别对它们的输入进行采样。

因此，为了让 D 触发器能在这段时间内正常工作， d 必须在 clk 由 0 变为 1 之前保持短时间的稳定——因此锁存器 A 中的 s 和 r 信号才能稳定，而且为了保持信号 s 和 r 的稳定，在 clk 变为 1 之后， d 也必须保持短时间的稳定。这是所需要的，因为信号 clk 和 $\overline{\text{clk}}$ 不是同时发生变化，这就导致了锁存器 A 经过一个延时（图 4-10）之后才能停止采样（“关门”）。在 clk 的 0-1 变化之前和之后 d 必须保持稳定的时间和分别被称为触发器的建立时间 (τ_{st}) 和保持时间 (τ_{ht})。对于一个下降沿触发器来说， d 必须在时钟 1-0 变化之前和之后保持稳定。

如下面两张图中所示，建立时间和保持时间的无法满足会让 D 触发器变得不稳定——众所周知的亚稳态情况。在图 4-13a 中，信号 d 发生变化与 clk 由 0 到 1 的变化是如此接近；因此，它破坏了触发器的建立时间，导致 $B.q$ 和 $B.\bar{q}$ 发生振荡。从另一方面讲，在图 4-13b 中，因为 d 在 clk 变为 1 之前进入了稳定状态，至少在时间上 $\geq \tau_{\text{st}}$ ，触发器正确地锁存 d ，使得 $B.q = d$ ， $B.\bar{q} = \bar{d}$ 。

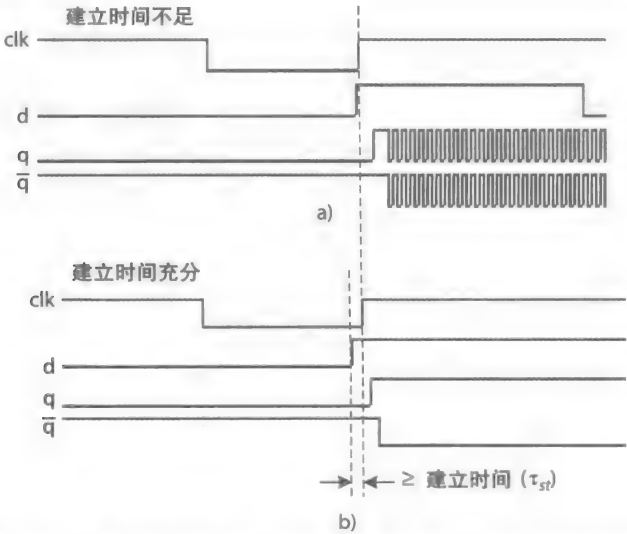


图 4-13 D 触发器时间：a) 建立时间不足；b) 建立时间充分

同样，在图 4-14a 中，当 clk 由 0 变为 1 之后， d 过早地发生变化，破坏了触发器所要求的保持时间。这同样会使信号 $B.q$ 和 $B.\bar{q}$ 产生振荡。从另一方面讲，在图 4-14b 中，在 clk 变为 1 之后 d 继续保持稳定，触发器工作正常，也使得 $B.q = d$ ， $B.\bar{q} = \bar{d}$ 。保持时间 (τ_{ht}) 是指在 clk 变为 1 之后 d 必须保持稳定的时间和。 τ_{ht} 必须大于或等于时钟到 q (τ_{cq}) 的延时，这是触发器能够稳定和输出最终结果 $B.q$ 和 $B.\bar{q}$ 所要求的时间。 τ_{cq} 也被称为时钟到输出的时间 (τ_{co})。

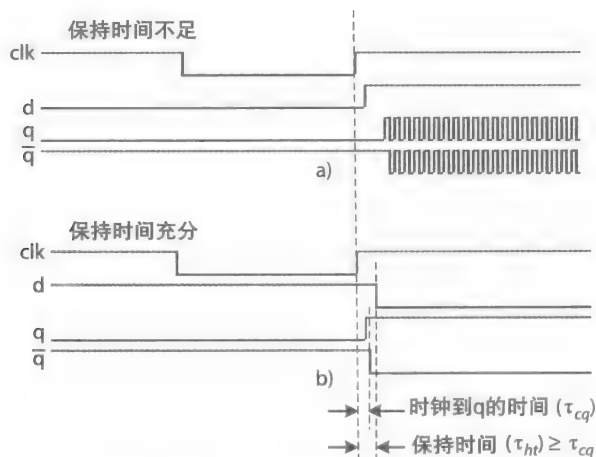


图 4-14 D 触发器时间：a) 保持时间不足；b) 保持时间充分

糟糕的保持时间

之前讨论过的与 d 和 clk 相关的信号源的建立和保持时间是在触发器内部，而不是在集成芯片的边缘或片内的一个模块。这种建立和保持时间也可以由芯片内或模块边缘的数据和时钟源决定。图 4-15 描述了一个片内触发器，在芯片边缘有 5 个接口信号，分别是 d 、 $clock$ 、 q ，还有低电平有效的 $_preset$ 和 $_reset$ 信号。就像图中所示，在这种情况下，这些输入信号在经过一些信号线路延迟之后将会影响到这个触发器。同样， q 作为这个触发器的输出，在经过芯片边缘的信号线路延迟之后将变为有效。另外，这些到达触发器的线路延迟有可能会不一样。因此，在这类情况下，时序图有可能与图 4-13 和图 4-14 中描述的都不一样。

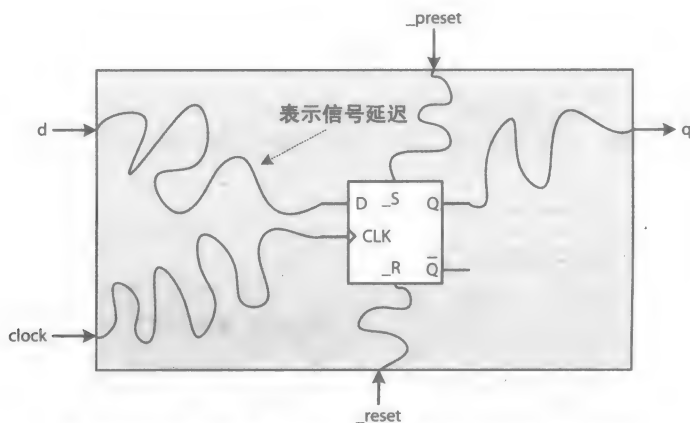


图 4-15 阐述在集成芯片内部或模块内部的信号延迟

图 4-16 描述了一个 D 触发器的时序仿真图，这个触发器是在 Altera 的现场可编程门阵列 (FPGA) 上进行了综合。所有的输入输出信号—— d 、 $reset$ 、 $preset$ 、 $clock$ 、 q 和 not_q ——都在 FPGA 的边缘，它们被相应地连接到 FPGA 内部触发器的输入输出上，并带有一定的延迟。就像图中所示，尽管最初时 $d = 1$ ，在 250.0ns (一个参考点) 处时钟上升沿之前的 2.902ns 时发生变化，变为了 0，触发器仍然可以取到 $d = 1$ ，然后在时钟上升沿之后的

5.922ns 处改变 q 的值为 1。在这种情况下，考虑到芯片边缘的输入信号，我们称这个触发器有一个糟糕的保持时间。触发器因为芯片的边缘也有一个 $\tau_{cq} = 5.922\text{ns}$ 的延迟。在图 4-17 中，这个触发器的仿真波形描述了相差 3.489ns 的糟糕的保持时间。

同样，如图 4-15 中所示，考虑到信号 d 和 clock 都在芯片的边缘，这个芯片会出现一个糟糕的建立时间也是有可能的。在这种情况下，考虑到时钟的 0-1 变化发生在芯片的边缘， d 的建立时间可能慢，但是仍然能够及时到达，满足触发器所要求的建立时间。 D 也有可能会连接到一个组合逻辑电路上，而且这个组合逻辑电路有一个输出连接到了这个触发器的 D 输入上。在这种情况下，信号 d 影响触发器时序的因素包括信号的延时和这个组合逻辑电路的传播延时。

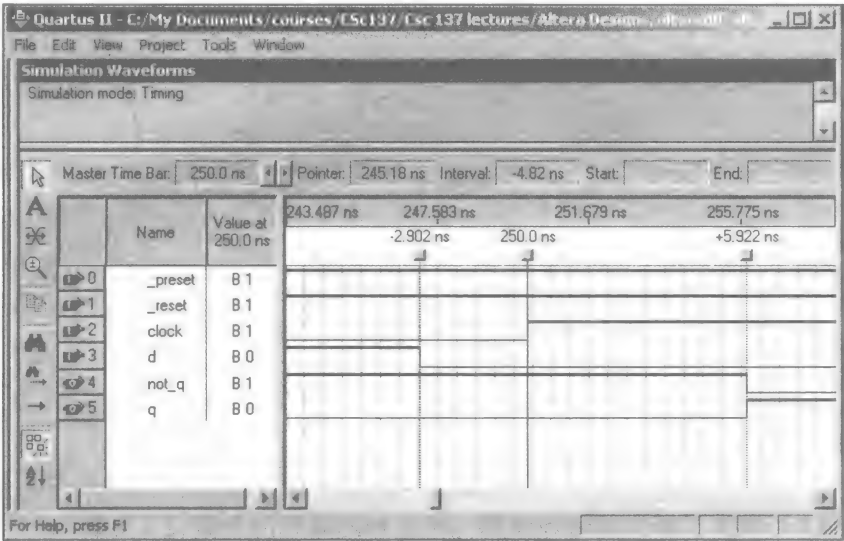


图 4-16 一个带有糟糕保持时间正常操作的 D 触发器时序图

154

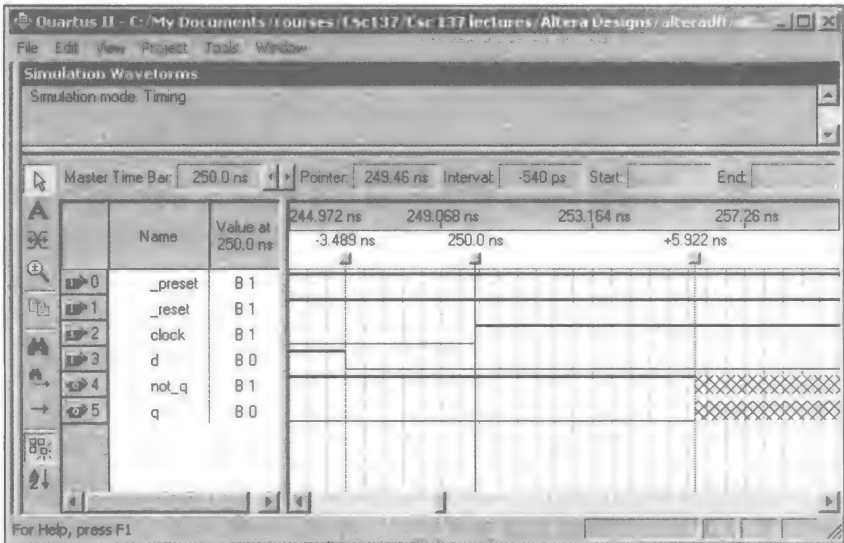


图 4-17 在 -3.489ns 处有糟糕的保持时间的 D 触发器时序图

而且, FPGA 或者专用集成芯片 (ASIC) 也可能包括固定和预制的电路模块。例如, 一个 FPGA 芯片包含了图 4-1 中的电路, 即预制的 CLA 加法器模块。另外, 这个模块可能包含一些内部时钟信号的延时。这样, 考虑到它们边缘的输入信号, 这些模块可能会有糟糕的建立时间或保持时间。我们已经知道, 当预制模块糟糕的建立 / 保持时间被 HDL 仿真软件模块化, 我们可以得到更精确的时序仿真图。

4.6 无相位差的时钟频率估计

图 4-18 中描述了操作 D 触发器所需的最小的时钟周期 (τ)。这个周期是时钟一次循环的时间, 它包括了时钟是 0 和时钟是 1 的时间总和。这个周期是由信号 d 的最大传播时延 ($\tau_{pd-\max}$), 时钟到 q 的最大时延 ($\tau_{cq-\max}$) 和触发器的建立时间 ($\tau_{st-\max}$) 计算而得。

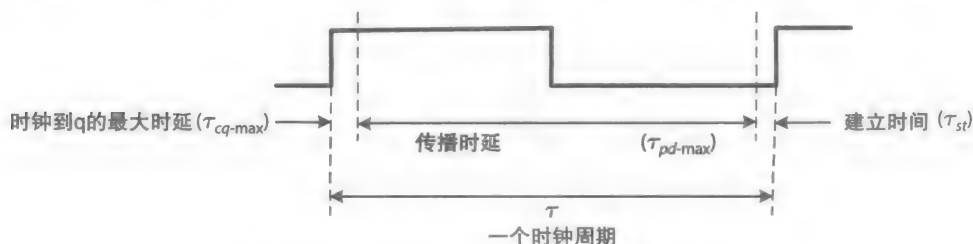


图 4-18 时钟周期与不同时延之间的关系

公式 (4-1) 用来计算估计的最小时间周期。这个估计的周期不考虑第 5 章介绍的与时钟相位差相关的一系列电路问题。

$$\tau \geq \tau_{cq-\max} + \tau_{pd-\max} + \tau_{st} \quad (4-1)$$

公式 (4-2) 描述了最大的时钟频率, 即每秒的时钟周期数目 (周期 / 秒), 也叫赫兹。它是 1 秒内时钟循环的次数。

$$f \leq \frac{1}{\tau} \text{ 周期/秒或赫兹} \quad (4-2)$$

典型地, 较大的频率值被称为千或千赫兹 (KHz); 兆或兆赫兹 (MHz); 千兆或吉赫兹 (GHz)。分别是指 1 秒钟内时钟周期的数目为 1 千次、1 百万次和 10 亿次。

4.7 触发器使能

一个典型的复杂的数字电路包括成千上万的触发器。所有这些触发器不是在同一时刻都要操作 (采样)。有些触发器是独立的操作, 也有些触发器是作为一个组同时被操作。例如, 一个有 16 个 32 位寄存器的处理器, 它含有 512 (16×32) 个触发器。我们说由一组 32 个触发器组成的 32 位寄存器将会在同一时刻被同时选中, 去存取一个新产生的 32 位的结果, 例如, 一个加法器执行一条加法指令产生的结果。因此, 在一个特定的时钟内, 一个用来选择一个触发器或一组触发器所需的额外的控制信号是必需的。

如图 4-19 描述了一个 D 触发器的设计, 并带有一个使能信号 e 用来控制一个 2-1 选择器。当 $e = 0$ 时, 选择器选择信号 q , 使得触发器重新加载 q 而保持原有的值。当 $e = 1$ 时, 选择器选择输入信号 d_in , 使得触发器加载信号, 改变 q 的值为 d_in 。我们说, 当 $e = 1$ 时这个触发器被使能或被选择; 当 $e = 0$ 时这个触发器被关掉或不被选择。

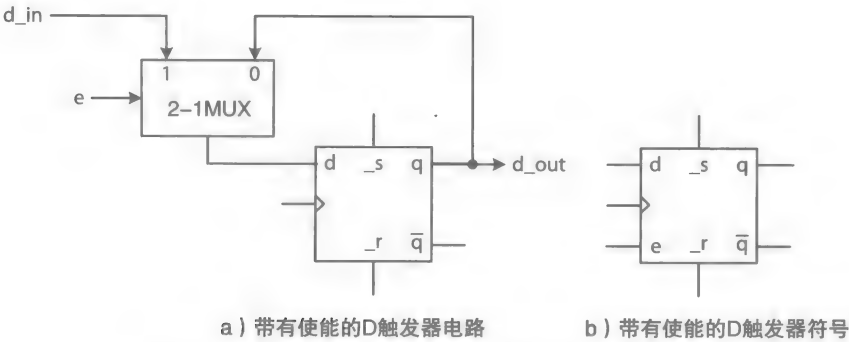


图 4-19 带有使能信号的 D 触发器电路及它的符号

156

4.8 其他触发器

历史中一种曾比较有名的触发器是 JK 触发器。如图 4-20a 中它有两个输入信号 j 和 k 。图中，信号 SET 和 CLR 分别表示高电平异步的预置和重置信号。当输入信号 j 和 k 都是 1 时，触发器的输出产生振荡，即每个时钟周期发生 q 变为 \bar{q} ， \bar{q} 变为 q 。JK 触发器的缺点是需要两个控制信号。然而，通常情况下使用 JK 触发器的电路有一个优点，相对于 D 触发器的 d 信号，信号 j 和 k 使用更简单的电路设计。一个边沿触发的 JK 触发器可以由一个 D 触发器设计而成，如图 4-21 所示。

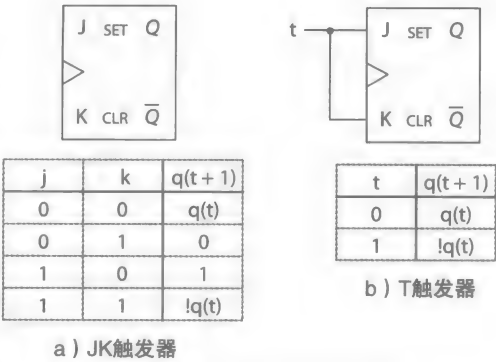


图 4-20 其他类型的触发器

图 4-20b 表示了另外一个被称为 T 触发器的触发器。这个触发器由一个 JK 触发器将其输入端 j 和 k 连到一个简单的信号 t 上设计而成，这个信号 t 代表了振荡。一个 T 触发器仅表现了两种功能：保持它的当前状态 (q) 或振荡。如果 $t = 0$ ，T 触发器保持它的值 q 。如果 $t = 1$ ，它的输出振荡：每个时钟周期内都会发生 q 变为 \bar{q} ， \bar{q} 变为 q 。T 触发器类似于 D 触发器，也有一个优点就是仅需要一个简单的输入信号 (t) 去同步地预置或重置它的 q 值。

通常情况下，任何触发器都可以系统地被其他触发器设计实现。例如，就像许多其他的时序电路，JK 触发器的动作可以由一个有限状态图 (FSD) 来表示。有限状态图和它的实现将会在第 5 章中介绍，即我们所知的有限状态机。

157

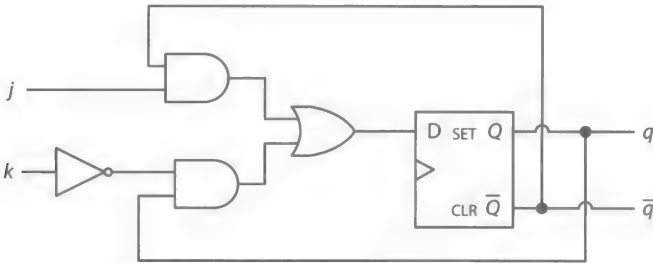


图 4-21 边沿触发的 JK 触发器的电路图

4.9 硬件描述语言模型

例 4-1 中用 Verilog 描述了一个高电平触发的 D 触发器行为模块，这个 D 触发器带有异步低电平触发的 `_reset` 和 `_preset` 信号。在锁存器和触发器中，非阻塞赋值或并行赋值运算符“`<=`”保证了同一个时钟下的多个锁存器和触发器的时钟同步。示例中“`always`”块的敏感信号表中有 4 个信号——`clock`、`_reset`、`_preset` 和 `d`。在这个描述中，`_reset` 被赋予了最高的优先级，`clock` 的优先级最低。当 `_reset = 1`（信号无效），`_preset = 1`（信号无效），`clock = 1` 时，只要 `clock` 的值一直保持为 1（采样电平），信号 `d` 的变化将会影响信号 `q` 的变化。这段 Verilog 代码在 `clock = 0` 时有一个缺省的“`else`”状态，暗示了当 `clock = 0` 时锁存器要保持它的 `q` 值不变。

例 4-1 一个高电平触发，异步低电平重置（`_reset`）和低电平预置（`_preset`）的 D 触发器行为模块、一个测试模块，还有仿真结果输出都在此一一列出。正如输出结果中所示，在仿真时间为 3 个单位时，`clock = 0`，`d = 1`，`q` 的值仍为 0，意味着锁存器如所想的那样保持了它的 0 状态。在时间为 4 个单位时，`clock = 1`，`d = 1`，`q` 在意料之中变为了 1。但是在时间为 5 个单位，`clock` 的值仍为 1 时，信号 `d` 变为了 0，此时 `q` 理所当然地变为了 0，阐释了锁存器的行为表现。

HDL 模型：

```
module d_latch
(
    input clock, _reset, _preset, d,
    output reg q,
    output nq
);

assign nq = ~q;
always@(clock or !_reset or !_preset or d)
begin
    if(!_reset)
        q <= 0;
    else if(!_preset)
        q <= 1;
    else if(clock)
        q <= d;
end
endmodule
```

测试模块：

```
'include "d_latch.v"
module tester();
reg clock, _reset, _preset, d;
wire q, nq;
d_latch dlatch1(clock, _reset, _preset, d, q, nq);
initial begin
$monitor("%4d clock = %b _reset = %b, _preset = %b d = %b q = %b nq = %b\n", $time, clock, _reset, _preset, d, q, nq);
end
initial begin
clock = 0; _reset = 1; _preset = 1;
#1 _reset = 0;
```

```
#1 _reset = 1;
#1 d = 1;
#1 clock = 1;
#1 d = 0;
#10 $finish;
end
endmodule
```

仿真输出：

```
Chronologic VCS simulator copyright 1991-2009
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12;
```

```
0 clock = 0 _reset = 1, _preset = 1      d = x q = x nq = x
1 clock = 0 _reset = 0, _preset = 1      d = x q = 0 nq = 1
2 clock = 0 _reset = 1, _preset = 1      d = x q = 0 nq = 1
3 clock = 0 _reset = 1, _preset = 1      d = 1 q = 0 nq = 1
4 clock = 1 _reset = 1, _preset = 1      d = 1 q = 1 nq = 0
5 clock = 1 _reset = 1, _preset = 1      d = 0 q = 0 nq = 1
```

```
$finish called from file "tester.v", line 21.
$finish at simulation time          15
```

VCS 仿真报告

Time: 15

CPU Time: 0.460 seconds; Data structure size: 0.0Mb

例 4-2 中描述了一个 Verilog D 触发器的行为模块。其中关键字 `posedge` 或 `negedge` 分别表示了上升沿或下降沿触发的动作。这个 Verilog 代码描述了一个上升沿触发的 D 触发器，并带有异步低电平触发的 `_reset` 和 `_preset` 信号，而且 `_reset` 信号有较高的优先级。这两个信号异步地控制电路，因为它们作为部分信号被列在“always”块的敏感信号表中；否则，如果它们没有出现在敏感信号表中，它们将仅在 `clk` 发生上升沿变化（在这种情况下）时才能同步地控制电路。

159

例 4.2 一个上升沿触发，带有异步低电平有效 `_reset` 和 `_preset` 信号而且有高电平使能 (*e*) 信号的 D 触发器行为模块、一个测试模块，还有仿真结果输出都在此一一列出。注意，在这个模块中，信号 *d* 和 *e* 没有包含在“always”块的敏感信号表中；如此一来，它们将会同步地影响 D 触发器。当 `_reset = 1`，`_preset = 1` 时，如果触发器已被使能 (*e* = 1)，时钟的 0-1 变化将会使触发器加载数据，使得 *q* = *d*。否则，如果 *e* = 0，触发器被关闭，*q* 的值将会保持不变。当时钟没有发生 0-1 变化时触发器 *q* 的值也会保持不变。也就是说，无论 `clk` 是 0 还是 1，信号 `_reset` 的 1-0 电平变化会异步地复位 *q* (*q* = 0)，而且信号 `_preset` 的 1-0 电平变化也会异步地置位 *q* (*q* = 1)。并且，并行的，非阻塞赋值符号“`<=`”使得多个实体的触发器（如果有）实现时钟同步。

如仿真里的输出所示，在仿真为 3 个单位时间 `clock = 0` 时，*d* 变为 1 并没有改变 *q* 的值，与所期望的一样。在时间为 4 个单位，`clock` 发生 0-1 变化时，*d* = 1 也如所期望的那样将 *q* 的值变为 1。在第 5 个时间单位，`clock` 仍然为 1，*d* 变为 0 时，*q* 如所想的那样没有发生变化，阐明了触发器的行为表现。

HDL 模块：

```
module dff
(
    input clock, _reset, _preset, d, e,
```

```

    output reg q,
    output nq
);

assign nq = ~q; //nq indicates not q
always@(posedge clock, negedge _reset, negedge _preset)
begin
    if(!_reset)
        q <= 0;
    else if(!_preset)
        q <= 1;
    else if(e)
        q <= d;
end
endmodule

```

测试模块:

```

`include "dff.v"
module tester();
reg clock, _reset, _preset, d, e;
wire q, nq;
dff dff1(clock, _reset, _preset, d, e, q, nq);
initial begin
$monitor("%4d clock = %b _reset = %b _preset = %b e = %b d = %b\n", $time, clock, _reset, _preset, e, d, q, nq);
clock = 0; _reset = 1; _preset = 1; e = 0;
#1 _reset = 0;
#1 _reset = 1;
#1 e = 1; d = 1;
#1 clock = 1;
#1 d = 0;
#10 $finish;
end
endmodule

```

仿真输出:

Chronologic VCS simulator copyright 1991-2009
 Contains Synopsys proprietary information.
 Compiler version D-2009.12; Runtime version D-2009.12;

```

0 clock = 0 _reset = 1 _preset = 1 e = 0    d = x q = x nq = x
1 clock = 0 _reset = 0 _preset = 1 e = 0    d = x q = 0 nq = 1
2 clock = 0 _reset = 1 _preset = 1 e = 0    d = x q = 0 nq = 1
3 clock = 0 _reset = 1 _preset = 1 e = 1    d = 1 q = 0 nq = 1
4 clock = 1 _reset = 1 _preset = 1 e = 1    d = 1 q = 1 nq = 0
5 clock = 1 _reset = 1 _preset = 1 e = 1    d = 0 q = 1 nq = 0

$finish called from file "tester.v", line 19.
$finish at simulation time          15

```

VCS 仿真报告

Time: 15

CPU Time: 0.460 seconds; Data structure size: 0.0Mb

参考文献

1. Mi-Sook Jang and Hoi-Jin Lee, "Methods of HDL simulation considering hard macro core with Negative Setup/Hold time," US Patent 7,213,222 B2, May 1, 2007.

练习

4.1 第一部分：图 4-22 所给电路图中， q_a 是电平触发的 D 锁存器的输出，在图 4-23 中为 q_a 完成时序波形。假设 $\tau_{st} = \tau_{cq} = 0$ 。

161

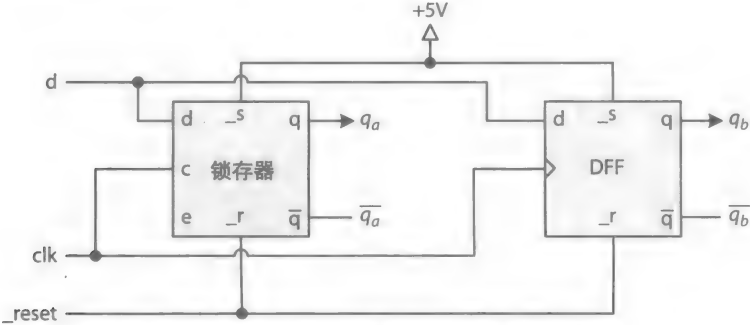


图 4-22 练习 4.1 和练习 4.2 的电路图

4.2 第二部分：图 4-22 所给电路图中， q_b 是上升沿触发的 D 触发器的输出，在图 4-23 中为 q_b 完成时序波形。假设 $\tau_{st} = \tau_{cq} = 0$ 。

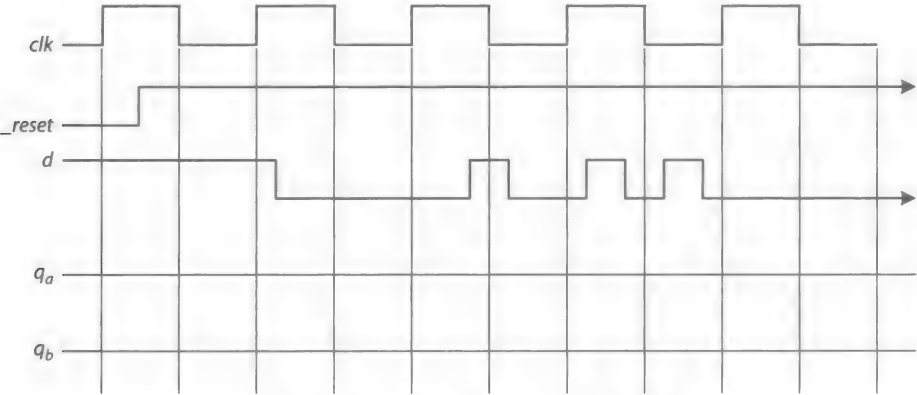


图 4-23 练习 4.1 ~ 练习 4.4 的波形图

4.3 第一部分：图 4-24 所给电路图中， q_a 是下降沿触发的 D 锁存器的输出，在图 4-23 中为 q_a 完成时序波形。假设 $\tau_{st} = \tau_{cq} = 0$ 。

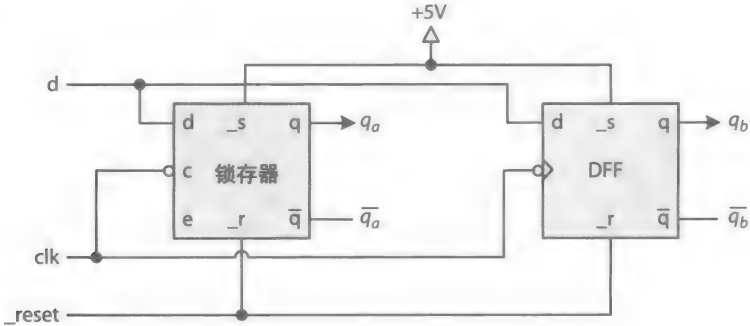


图 4-24 练习 4.3 和练习 4.4 的电路图

162

4.4 第二部分: 图 4-24 所给电路图中, q_b 是下降沿触发的 D 触发器的输出, 在图 4-23 中为 q_b 完成时序波形。假设 $\tau_{st} = \tau_{cq} = 0$ 。

4.5 图 4-22 所给电路图中, q_b 是上升沿触发的 D 触发器的输出, 在图 4-25 中为 q_b 完成时序波形。假设 $\tau_{st} = 0.15\text{ns}$, $\tau_{cq} = 0.1\text{ns}$ 。

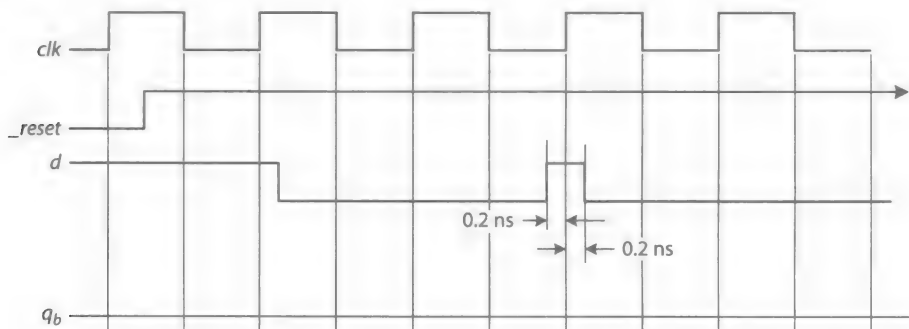


图 4-25 练习 4.5 的时序波形图

4.6 图 4-22 所给电路图中, q_b 是上升沿触发的 D 触发器的输出, 在图 4-26 中为 q_b 完成时序波形。假设 $\tau_{st} = 0.1\text{ns}$, $\tau_{ht} = 0.1\text{ns}$, $\tau_{cq} = 0.2\text{ns}$ 。

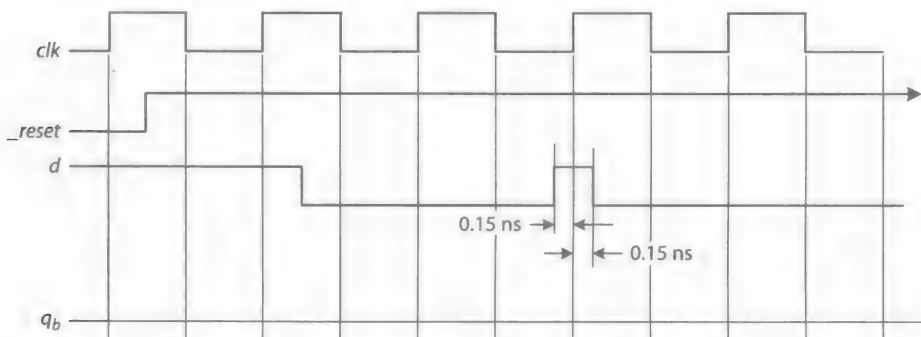


图 4-26 练习 4.6 的波形图

4.7 已知一个 D 触发器的建立时间是 0.1ns , 保持时间是 0.1ns , d 最大的传播时延是 0.3ns , 请确定 D 触发器能够正常工作时的时钟最大频率。

4.8 请用与非门实现一个带有重置信号和预置信号的 D 触发器的 Verilog 结构模型, 并对它进行仿真。

时序电路：小型设计

5.1 简介

类似于组合电路（CC），时序电路也可以分为小型电路和大型电路。例如，输出序列数 0, 1, 2, 3, … 的计数器是一个小型时序电路，而可以同时执行多条指令的处理器（CPU）是一个大型时序电路。每条指令都需要一组操作来完成，其中涉及一个或多个寄存器，甚至是存储器。

所有的小型 and 大型电路都由触发器和一系列的组合电路组合而成，例如图 5-1 中所示的 2 位的计数器。这个计数器由两个触发器和一系列包含 2 位加法器的组合电路组合而成。相对于组合电路，时序电路包含状态和传输，在每个时钟周期内可以从一个当前状态转换到另外一个状态。当前的状态由触发器 q 位的状态决定。在这个图中，如果两位的 $Q = q_1q_0 = (00)_2$ ，那么我们就说这个计数器当前的状态是 0；如果两位的 $Q = q_1q_0 = (01)_2$ ，那么我们就说这个触发器当前的状态是 1，等等。计数器每个时钟周期也会输出 2 位的状态值 $Q = q_1q_0$ 作为计数。

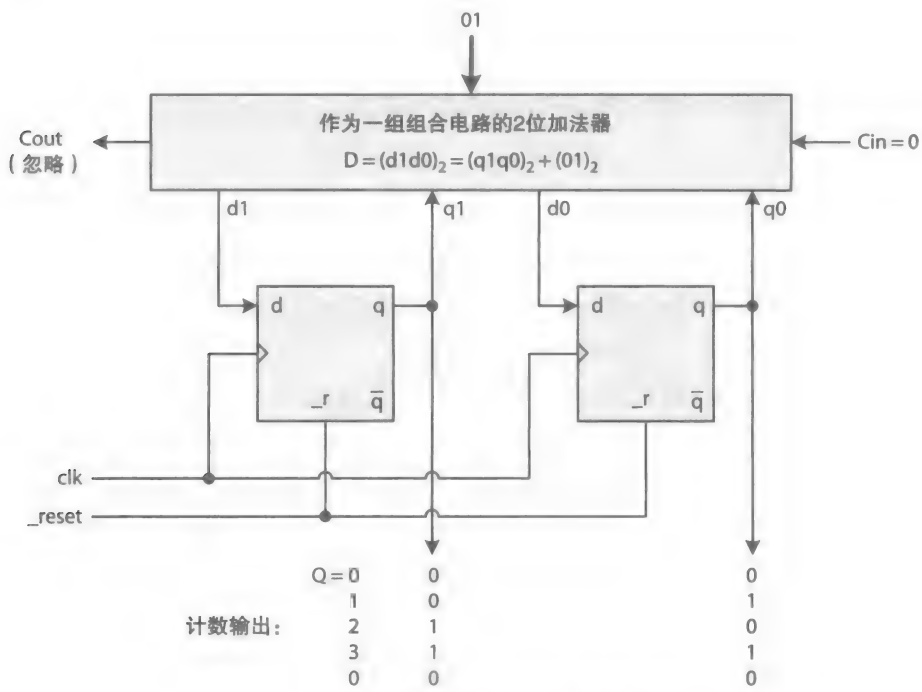


图 5-1 带有两个触发器的 2 位计数器的 FSM

加法器（计数器组合电路的集合）产生计数器的下一个状态作为一个 2 位的数字 $D = d_1d_0 = q_1q_0 + 1$ 。如果这个计数器的当前状态是 $Q = q_1q_0 = (00)_2$ ，那么它的下一个状态是 $D =$

$d_1d_0 = (01)_2$ 。 d_1 和 d_0 在下一个时钟周期内被保存。

这种时序电路的设计需要额外的方法学。时序电路的设计问题通常是建模为一个有限状态图 (FSD)。FSD 由环形的状态和弧 (箭头) 的转换组合而成。它正式指定了目标时序电路的行为。这个 2 位的计数器有 4 个状态, 分别记为 0、1、2、3。它从状态 0 转换为 1, 再从状态 1 转换为 2, 然后再从 2 转换为 3, 最后再从状态 3 回到 0。有限状态图系统转换成一个叫作有限状态机 (FSM) 的电路, 如图 5-1 所示。

大型时序电路的设计问题通常可划分为数据通路和控制单元的设计。数据通路包括 CC 模块, 比如算术逻辑单元 (ALU)、多路选择器 (MUX), 还有译码器和小型的时序电路, 比如寄存器和计数器。大型时序电路的设计将会在第 6 章中进行介绍, 特别地, CPU 的设计将会在第 8 章中进行介绍。

有时, 设计时序电路时不优先构造有限状态图也是可行的。在很多情况下, 简化一些小型和大型时序电路的设计是一个很重要的理念, 包括 CPU。例如, 图 5-1 中的组合电路 (也就是加法器) 就是一个例外, 计数器作为一个已知的功能, 不需要 FSD 就可以设计出来。

另一方面, 比如序列识别器的设计, 输入为一系列的 1 和 0, 一旦识别出一组特定的序列输入就输出 1, 这个设计的实现需要 FSD。这种情况与示例中 2 位的计数器形成对比, 因为让一个设计者提前知道序列识别器的组合电路会表现出什么特别的功能几乎是不可能的。

时序电路也很容易受到硬件电路的影响, 比如发生在传输时的瞬时故障, 这改变了时序电路的状态, 导致了失灵。使时序电路不受硬件环境干扰的一种方式是使用容错设计。

这一章我们将以一个小型状态机设计问题开篇, 然后说明通常情况下状态机的设计实现, 包括容错 FSM 设计。我们也将说明时序电路的时序要求, 一起讨论用 Verilog 实现的状态机的实例。

5.2 状态机介绍: 寄存器设计

寄存器作为一个小型时序电路, 被用作存储模块来保存组合电路的输出。作为一个移位寄存器, 要有能力将它本身的数据向左或向右移动一定的位数。我们选择一个简单的寄存器作为第一个设计问题有以下几点原因:

- 为了说明设计划分问题。
- 为了首先设计一个简单的状态机。
- 为第 4 章用到的触发器提供一个典型的设计。

图 5-2 描述了一个带有输入 $X = x_3 \cdots x_0$ 和输出 $Z = z_3 \cdots z_0$ 的 4 位并行加载寄存器。假设 $_reset = 1$ (即无效), 如果寄存器已被使能 (就是 $enable = 1$), 那么寄存器将在下一个 clk 的上升沿加载 X , 使得 $Z = X$ 。否则, 如果 $enable = 0$, 寄存器将不被使能, 继续保持它的值不变。

在这个图中, 假设每一个触发器的异步预置信号 $_s$ 是 1 (无效), 这些在图中没有展现出来。然而, 如果让 $_reset$ 信号初始化一些寄存器的位为 1 而其他的位为 0, 那么 $_reset$ 信号应当连接到某些触发器的输入端 $_s$ 上来把它们初始化为 1, 并且连到其他触发器的输入端 $_r$ 上。所有未用到的输入端 $_s$ 和 $_r$ 应当被置为无效, 在这个图中是低电平有效的话, 它们应当被连到 1 上。

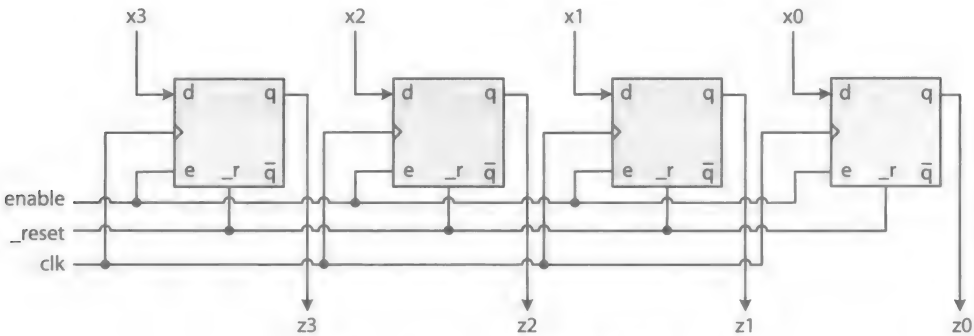


图 5-2 一个 4 位并行加载寄存器

5.2.1 寄存器模型

图 5-2 中的 $n = 4$ ，一个 n 位寄存器的设计可以由 n 个 1 位的寄存器片组合而成。如图 5-3 所示，这个寄存器片的行为可以被典型地建模为一个有限状态图。通过分析设计的问题可知，这个电路表现出一系列特定而又有明确数量的状态。因为这个 1 位寄存器存储 0 或 1 作为它的内容，它的状态图表现出两个可能的状态（也就是两个圈）。这些弧代表了从一个状态转换到另一个状态，包含每一种转换情况。

如图中所示，如果这个寄存器如所想的一样没有被使能（也就是 $e = 0$ ），寄存器将保持它的当前状态 0 或 1。这意味着有一个弧（弧 1）会从状态 0 回到它本身，也有另外一个弧（弧 4）会从状态 1 回到它本身。与每一次转换相关的信号值都在弧上列了出来。例如，与弧 1 和弧 4 相关的信号是 $e = 0$ ， x 的值不产生影响（0 和 1）。如果这个寄存器片的状态是 0，而且此时 $e = 1$ ， $x = 1$ ，那么在下个时钟周期这个寄存器片将从状态 0 转换到状态 1，即弧 3 所示。其他的弧也都类似。

图 5-4 中描述了一个 1 位寄存器片作为一个 FSM 的详细框图。它包括一个简单的触发器用来保存这个 1 位寄存器的状态是 0 还是 1，还包含了一个组合电路来输入当前的状态，即 q 信号所示，另外还有其他的输入信号 x 和 e ，用来确定下一个状态的输出，即信号 d 所示。这个寄存器片的输出被定义为 $z = q$ 。

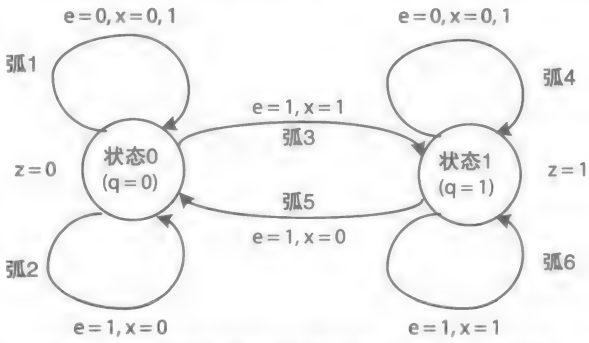


图 5-3 一位寄存器片被建模为一个有限状态图

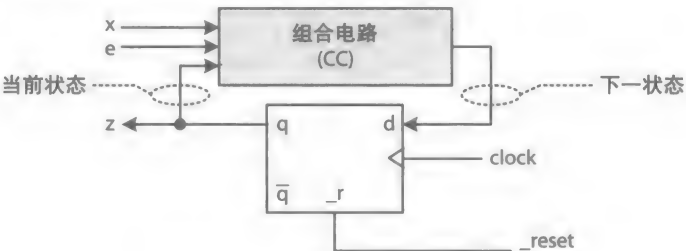


图 5-4 1 位寄存器片的详细框图

165
167

d 信号的逻辑表达可以由一个真值表确定，也叫**转换表**。这个表是有限状态图中信息的表格方式的表达。表 5-1 是 1 位寄存器片的转换表。例如，表中的前两行表示弧 1（图 5-3），第 3 行表示弧 2，第 4 行表示弧 3，等等。从这个转换表中可以得出表达式 $d = \bar{e}q + ex$ 的最小值。

168

表 5-1 1 位寄存器片的 FSD 所对应的转换（真值）表

| 当前状态 | 外部输入 | | 下一状态 |
|------|------|-----|------|
| q | e | x | d |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

图 5-5 表示一个 1 位寄存器片的电路，注意这里的表示中有一个 2-1 选择器，如果 $e = 0$ ，那么 $d = q$ ，相反如果 $e = 1$ ，则 $d = x$ 。回忆起在第 4 章中有一个相同的电路，是一个带有使能信号的触发器（图 4-19）。而这里，这个电路是使用有限状态图来设计的。

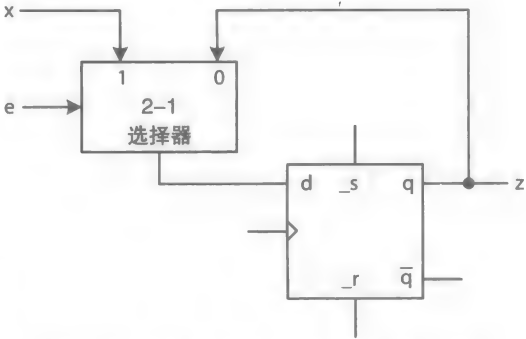


图 5-5 1 位寄存器片（与图 4-19 一样）

5.2.2 多功能寄存器

有时候，寄存器需要表现出多个功能中的一种。图 5-6 是一个带有 n 位输入 X 、 n 位输出 Z 和 2 位功能码 $F = f_1f_0$ 的 4 种功能的寄存器框图。当寄存器的高电平有效的 enable 信号有效时，该寄存器被使能。该寄存器还需要一个低电平有效的 $_reset$ 信号来异步地将它初始化为 0。这个寄存器的 4 种功能分别是同步复位（清零）、并行加载、算术右移并复制符号位和右移并从左边进 0。

169

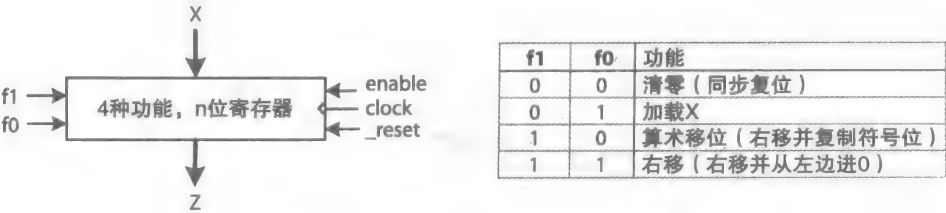


图 5-6 一个 4 种功能的寄存器框图

1. 位串行设计

图 5-7 是 4 种功能 1 位寄存器片的详细框图和有限状态图。和之前的示例形成对比，这里假设这种情况中的触发器包含使能信号 e 。而且，为了清晰与方便，一些不影响的信号没

有表示出来，在有限状态图的转换动作中也省略了。例如， i 和 x 在条件 $F = 0$ ($f_1 f_0 = 00$) 发生时是不产生影响的信号，因为寄存器片必须被同步地初始化为 0，而不关心输入信号 i 和 x 的值，那么在 F 条件发生时它们不再被表示出来。

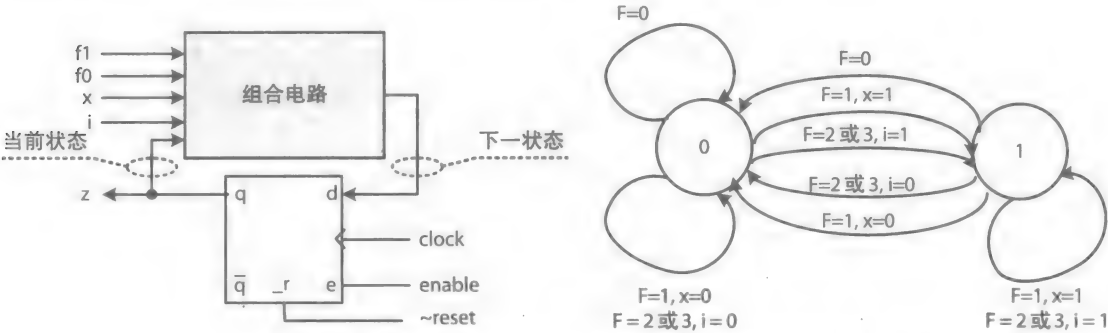


图 5-7 4 种功能 1 位寄存器片的详细框图和有限状态图

在图 5-7 中，这个组合电路有 5 个输入端—— q （当前状态）、 f_1 、 f_0 、 i 和 x ，还有一个输出端 d （下一状态）。它的真值表将会包括 32 行，因此这里就不再表示了。使用 Espresso 最小化软件得出表达式 $d = \bar{f}_1 \bar{f}_0 x + f_1 i$ 。4 种功能 1 位寄存器片的最终电路和符号在图 5-8 中表示了出来。使用 4 个同样的寄存器片可以得到 4 种功能的 4 位寄存器，即图 5-9 中所示。图中的每个寄存器片，除了最左边的那个，其输入端 i 都连到了前面所连寄存器片的输出端 z 。最左边寄存器片的输入端 i 被定义为 $i = f_1 \bar{f}_0 z$ 。当 $F = 2$ 时它将复制 z_3 的值，对外表现出算术右移。

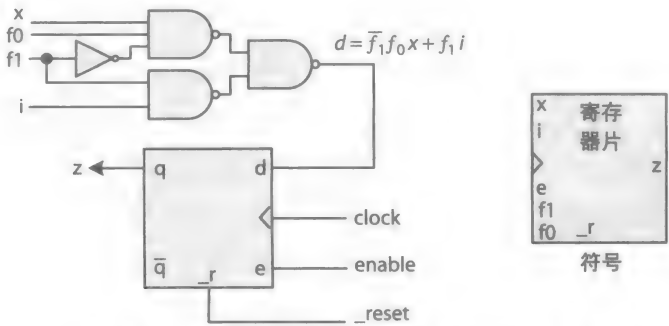


图 5-8 4 种功能的 1 位寄存器片和它的符号

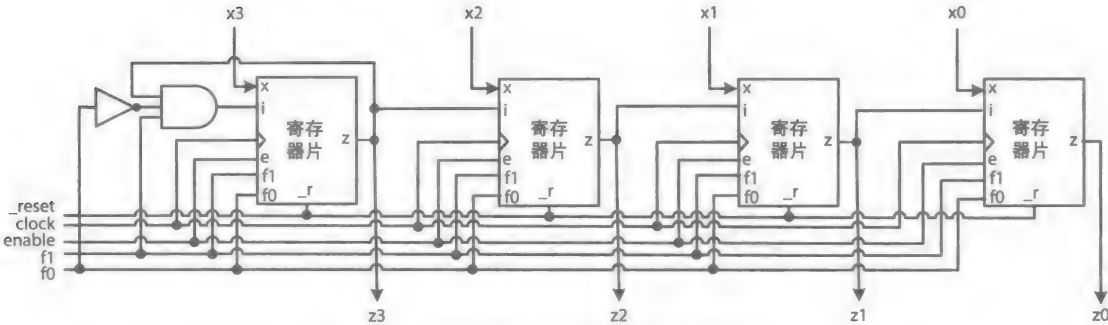


图 5-9 4 种功能的 4 位串行寄存器设计

注意图 5-10, 有一个在图 5-8 中定义的带有输入端 0、x、i 和 i (i 信号用到了两次) 的 1 位 4-1 选择器。选择器 SOP 的表达式简化为 $d = \bar{f}_1 \bar{f}_0 x + f_1 i$, 这和图 5-8 中用有限状态图得到的是一样的。这是在不需 FSM 模型的情况下去设计位串行或位并行有限状态图的科学有效的方式。然而, 使用这种技术需要分析设计问题, 确定是否有已知的可用的组合电路模块, 比如这种情况中的选择器就可以在这种设计中使用。这在使用位并行技术设计 4 位的 4 种功能的寄存器中将再次介绍。然而, 在分析设计问题时如果不能确定一个已知的模块, 则必须使用有限状态图对这个设计问题进行建模。

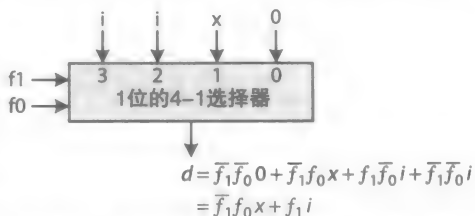


图 5-10 带有输入信号 0、x 和 i (用到两次) 的 1 位 4-1 选择器

2. 位并行设计

图 5-11 描述了图 5-6 中所给的 4 种功能寄存器的设计, 使用了一个 4 位并行加载寄存器和一个 4 位的 4-1 选择器。这个选择器的 4 个 4 位输入中的一个连到了它的输出上, 这个值然后被加载到一个 4 位并行寄存器中。这个选择器的 4 个输入定义如下:

- 输入 0: 同步清零 ($F = 0$), 连接到地, 置为逻辑 0。
- 输入 1: 并行加载 ($F = 1$), 连到 4 位的输入 X 上。
- 输入 2: 算术右移 ($F = 2$), 连到 $\{z_3, z_3 z_2 z_1\}$, 符号位 z_3 和寄存器的输出 Z 的高三位拼接起来 (由 {} 表示出来) 生成一个 4 位的数。
- 输入 3: 右移 ($F = 3$), 连到 $\{0, z_3 z_2 z_1\}$, 0 和寄存器的输出 Z 的高三位拼接起来生成一个 4 位的数。

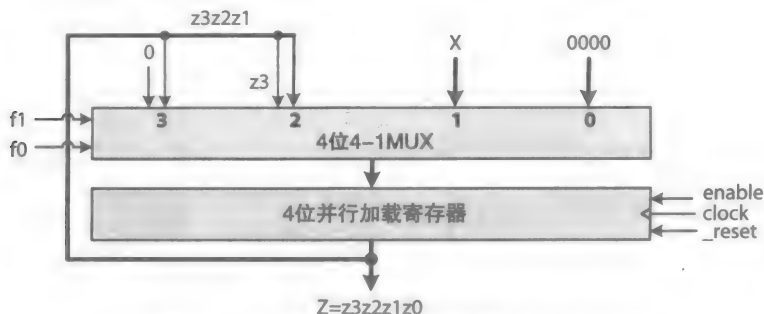


图 5-11 4 位的 4 种功能寄存器的位并行设计

5.3 FSM 设计

在前面的章节中已经列举过位串行和位并行 FSM 的简单实例。通常情况下, FSM 按种类可分为 Mealy 状态机、Moore 状态机和混合状态机。另外, FSM 的组合电路可分为两部分: 一部分表现为下一个状态发生器 (NSG), 另一部分表现为输出信号发生器 (OG)。NSG 决定了下一个状态, OG 产生输出的信号。

图 5-12 是一个 FSM 的详细框图。如果 FSM 的输出由它当前的状态或它当前的输入决定, 而不是由时钟计数、复位、直接连到触发器上的置位信号和触发器上的使能信号确定, 那么这种输出被称为 Mealy 输出, 这个 FSM 被称为 Mealy 状态机。外部输入中的一个或多个

个值的改变将会使得 Mealy 输出发生变化，而与时钟信号无关。在图中，这将表现为一根粗虚线，连接着外部输入信号和输出信号发生器的输入端。Mealy 输出异步地取决于外部输入的值。Moore 输出不像 Mealy 输出，它同步地取决于外部输入的值。混合状态机同时包括 Mealy 输出和 Moore 输出。

172

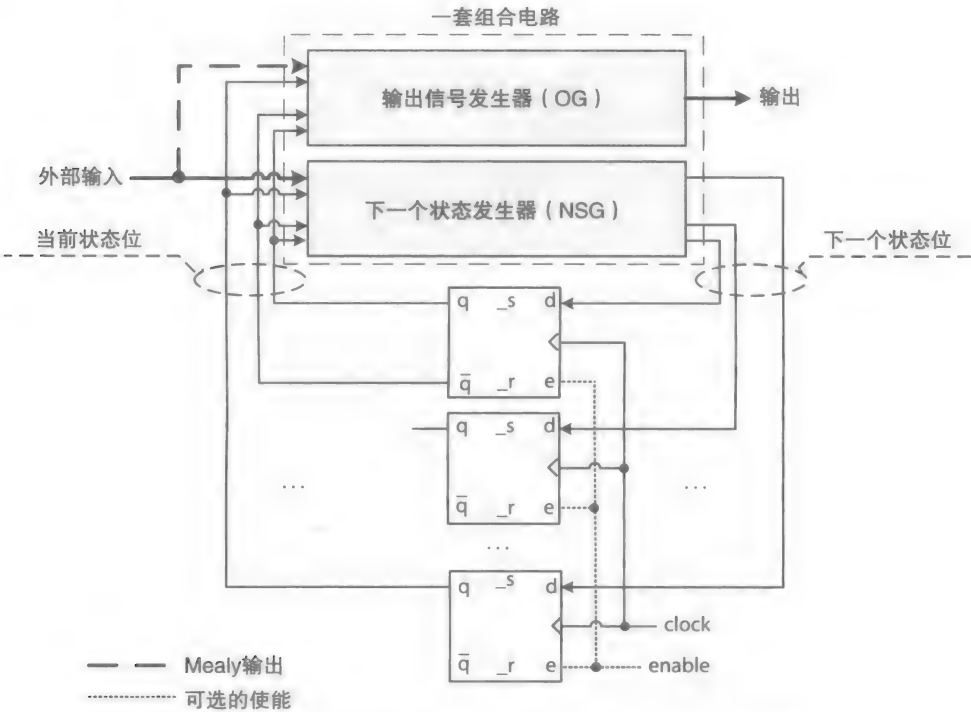


图 5-12 阐述 Mealy 状态机和 Moore 状态机的详细框图

我们使用一个序列识别器作为实例来继续讨论这个话题，还有影响电路尺寸的设计选择。序列识别器工作起来更像组合电路数字锁的控制器。它一次监视一位输入，而且识别器每次检测到目标序列都会输出 1，例如 3 位的序列“101”。识别器可以识别重叠或不重叠的序列。例如，输入序列“10101”包含两个重叠的序列“101”，其中输入序列中间的“1”是共用的。从另一方面讲，输入序列“101101”包含两个不重叠的“101”序列。

例 5-1 检测重叠序列“101”的 Moore FSM 的设计。

解：图 5-13 所示为带有输入 x 和输出 z 的序列识别器的顶层模块框图。它的 Moore 有限状态图也由所列 A、B、C 和 D 四个状态表示了出来。高电平有效的 reset 信号用来异步地将状态机初始为状态 A，即有限状态图中的 reset 箭头所示。输入的序列一次处理一位。序列识别器每次处理目标序列的下一位时将会发生转换，进入一个新的状态。例如，如果识别器正在状态 C 中，意味着它已经接收了目标序列的第一个两位。每个状态的 z 输出在下面表示了出，而且当识别器接收到目标序列的最后一位并进入状态 D 时 z 变为 1。在其他状态中 z 的值都是 0。识别器每次都会拒绝所输入的所有其他情况的 3 位序列，并重新开始。下面将介绍其他解决方案的细节。

173

如果有限状态图中的每个状态都有一个独特的转换（弧），那么这个有限状态图是确定的。然而，比如说图 5-13 中的状态 A，当 $x = 1$ 时还有第二个转换到状态 C，那么这个有限

状态图是不确定的。在这种情况下，当 $x = 1$ 时，将不能确定应当从状态 A 转换到 B 还是转换到 C。这里有两种方式来实现确定的有限状态图：

1) 二进制编码状态——在这种情况下，这些状态都由唯一的最小可能位数的二进制数来标识（标记）。例如，图 5-13 中的 4 个状态可以由 2 位的二进制数来标识：00、01、10 和 11。

2) 独热码状态——在这种情况下，这些状态均被一个唯一的二进制数所标识，每个状态仅由一位 1 组成（独热），其他位均保持 0。例如，图 5-13 中的 4 种状态可以由 4 位的独热码 0001、0010、0100 和 1000 来标识。

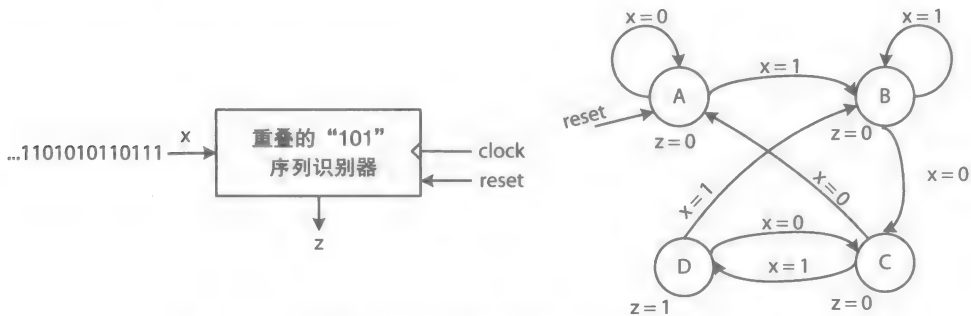


图 5-13 “101”序列识别器的框图以及它的 Moore FSD

5.3.1 二进制编码状态

有限状态图中对状态进行编码所需的最小位数由公式 (5-1) 确定，其中 k 是状态的总个数。符号 $\lceil \cdot \rceil$ 表示上取整运算。

$$\text{位数} = \lceil \log_2(k) \rceil \quad (5-1)$$

例如，如果状态的数量超过了 4 而且低于 8（即 $4 < k < 8$ ），那么 5 ~ 8 个状态都需要 3 位的数来表示。哪一个数字被分配到哪一个状态是一个逻辑优化问题。图 5-14 用二进制编码状态表示了图 5-13 中序列识别器的详细框图。

174

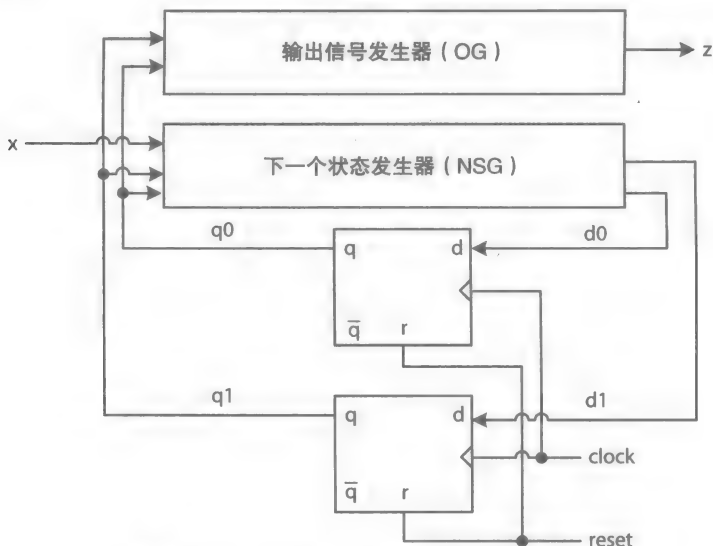


图 5-14 “101”序列识别器的详细框图

接下来的两个转换（真值）表（表 5-2 和表 5-3）是由有限状态图确定的，其中二进制数 00 用来表示状态 A，01 表示状态 B，10 表示状态 C，11 表示状态 D。真值表用来确定与每个状态变量 d_1 、 d_0 和输出变量 z 相关的最小 SOP 表达式（公式（5-2））。图 5-15 是带有上升沿触发的触发器和高电平有效的复位信号的完整电路。

表 5-2 由图 5-13 中有限状态图得出的下一个状态转换表

| | 当前状态 | | 输 入 | 下一个状态 | | |
|---|-------|-------|-----|-------|-------|---|
| | q_1 | q_0 | x | d_1 | d_0 | |
| A | 0 | 0 | 0 | 0 | 0 | A |
| | 0 | 0 | 1 | 0 | 1 | B |
| B | 0 | 1 | 0 | 1 | 0 | C |
| | 0 | 1 | 1 | 0 | 1 | B |
| C | 1 | 0 | 0 | 0 | 0 | A |
| | 1 | 0 | 1 | 1 | 1 | D |
| D | 1 | 1 | 0 | 1 | 0 | C |
| | 1 | 1 | 1 | 0 | 1 | B |

表 5-3 由图 5-13 中有限状态图得出的输出信号发生器的真值表

| 当前状态 | | 输 出 |
|-------|-------|-----|
| q_1 | q_0 | z |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

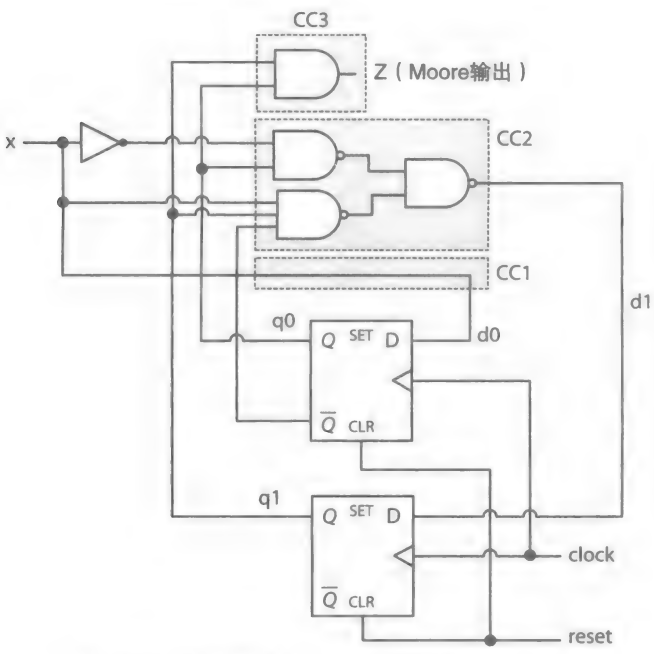


图 5-15 检测重叠序列“101”的 Moore FSM

$$\begin{aligned}d_1 &= q_0 + \overline{q_1}x = (\overline{q_0})(\overline{q_1x}) \\d_0 &= x \\z &= \overline{q_1}q_0\end{aligned}$$

(5-3)

177

5.3.2 独热码状态

二进制编码状态的设计技术可以减少触发器的数量，而独热码状态的设计技术可以减小组合电路的大小。独热码在可编程的逻辑设备（PLD）中显得很有优势，PLD 中有大量的可配置的逻辑块（CLB），每个 CLB 中有一个或多个触发器。例如，一个 FPGA 中有成千上万个 CLB，会包含成千上万个触发器。因此，使用更多的触发器（每个状态一个）和更少的复杂的组合电路将会更有效。独热码设计技术更像是为了生产传播时延更小的电路，其中不计入传播线的时延，这在某些 PLD 设计中可能会更长一些。

在每个时钟周期中，仅有一个触发器会被置位（即 q 为 1），其他的触发器保持复位（即 q 为 0）。例如，假如不使用图 5-13 中用 2 位数去标识 4 个状态的方式，而使用 4 位的独热码 0001、0010、0100 和 1000 来表示。图 5-18 是相应的独热码 FSM 的详细框图。注意，复位之后，状态机必须从状态 A($q_3q_2q_1q_0 = 0001$) 开始，同时只有一个触发器被置位。因此，如图中所示，复位信号必须连到与 q_0 相关的触发器的置位（ s ）输入端，同时连到其他触发器的复位（ r ）输入端。表 5-4 是下个状态发生器的真值表，表 5-5 是输出发生器的真值表。没有的表项被视为不关心项，不再列出。然而为了进一步简化电路，这些不关心的输出值在卡诺图、Espresso 文件和 HDL 建模中应当被列出来。另外，为了在 HDL 建模中避免产生隐式的锁存器，不关心的值会变得尤其重要。

178

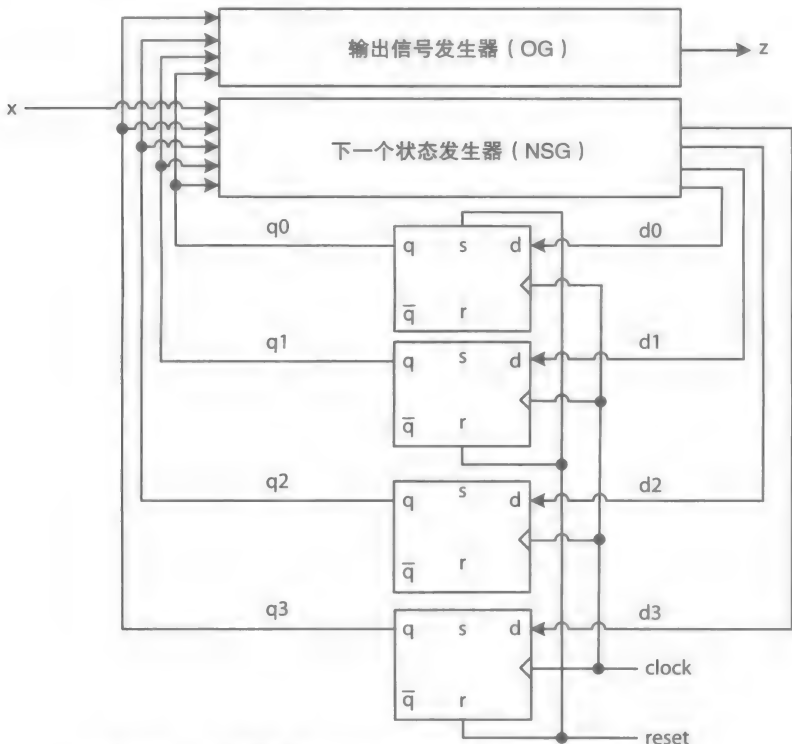


图 5-18 检测序列“101”的独热码 FSM 的详细框图

表 5-4 图 5-12 中 FSD 所对应的独热码设计中下一个状态发生器的真值表

| 当前状态 | | | | | 输 入 | 下一个状态 | | | | |
|-------------------------------------|---|---|---|---|-----|-------|-------|-------|-------|---|
| $q_3 \quad q_2 \quad q_1 \quad q_0$ | | | | | x | d_3 | d_2 | d_1 | d_0 | |
| A | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | A |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | B |
| B | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | C |
| | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | B |
| C | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | A |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | D |
| D | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | C |
| | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | B |

表 5-5 图 5-12 中 FSD 所对应的独热码设计中输出发生器的真值表

| 当前状态 | | | | | 输 出 |
|-------------------------------------|---|---|---|---|-----|
| $q_3 \quad q_2 \quad q_1 \quad q_0$ | | | | | z |
| A | 0 | 0 | 0 | 1 | 0 |
| B | 0 | 0 | 1 | 0 | 0 |
| C | 0 | 1 | 0 | 0 | 0 |
| D | 1 | 0 | 0 | 0 | 1 |

公式 (5-4) 列出了下个状态变量 d_3 到 d_0 和输出变量 z 的最小表达式。

$$\begin{aligned}d_3 &= q_2 x \\d_2 &= \overline{q_2} \overline{q_0} \overline{x} \\d_1 &= \overline{q_2} x \\d_0 &= \overline{q_3} \overline{q_1} \overline{x} \\z &= q_3\end{aligned}$$

(5-4)

179

图 5-19 表示最终的电路。与图 5-17 中的二进制编码 FSM 形成对比，独热码 FSM 需要更多的门电路，但是每一个下个状态位需要更少的电路，输出 z 不需要电路。

例 5-2 检测重叠序列“101”的 Mealy FSM 的设计。

解：图 5-20 是一个序列识别器的 Mealy 有限状态图的框图。注意，在这种情况下， z 作为 Mealy 输出只与弧相关，而不是状态。每个弧的标签由两部分组成，由一个斜杠 (/) 隔开，输入 (这里只有 x) 在斜杠的左边，输出 (这里只有 z) 在斜杠的右边。另外，在介绍 Mealy 解决方案之前，下面先通过这个实例对 Mealy 和 Moore 的设计问题进行讨论。

图 5-21 为 Mealy FSM 的详细框图。注意，如果 FSM 在状态 C 中，那么当 $x = 0$ 时 $z = 0$ 或者当 $x = 1$ 时 $z = 1$ 。因此，不出所料，Mealy 输出 z 异步地取决于外部输入 x 。这是 Mealy FSM 的典型表现。一旦 x 信号发生变化， z 信号也会变化，而与时钟信号无关。这个有限状态图有 3 种状态。它可以使用两个触发器通过二进制编码状态标识来实现，也可以使用三个触发器通过独热状态码来标识实现。

假如状态 A、B 和 C 通过 2 位二进制编码来表示，4 种可能的二进制状态中有一个将不

被使用。例如，假设二进制标识 00 用来表示状态 A，01 表示 B，10 表示 C。那么二进制标识 11 就没有用到，而且构成了一种未知的未定义的机器状态。环境因素（比如瞬时故障）造成的一个或多个触发器状态的变化同时会造成 FSM 的状态变化。例如， q_1 或 q_0 中某一位的异常变化将会把状态 01(C) 变为已知的状态 00(A) 或未知的状态 11(比如 D)。通常情况下，在设计中会有很多种办法去处理未知的状态。

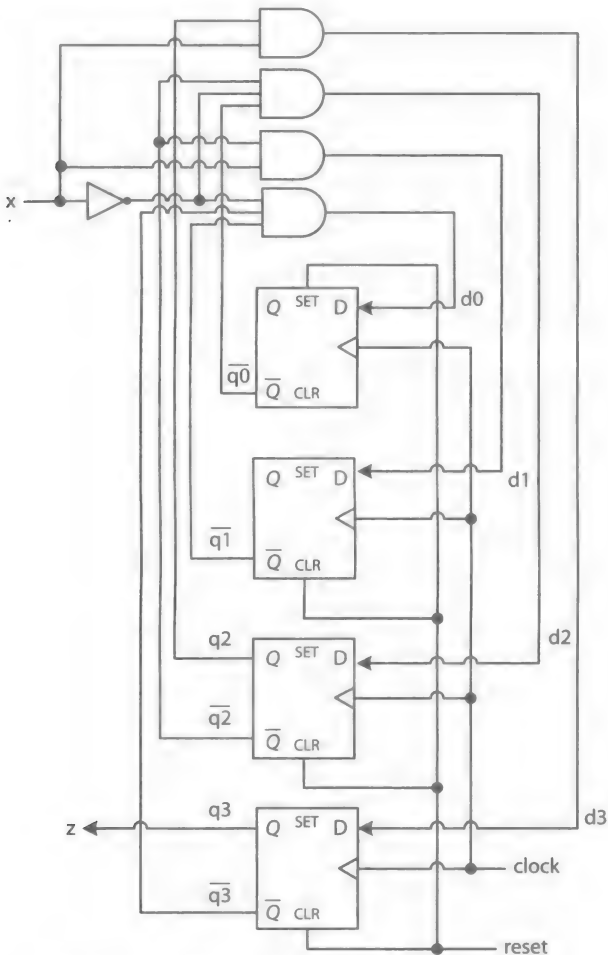


图 5-19 检测序列“101”的独热码设计 FSM

180

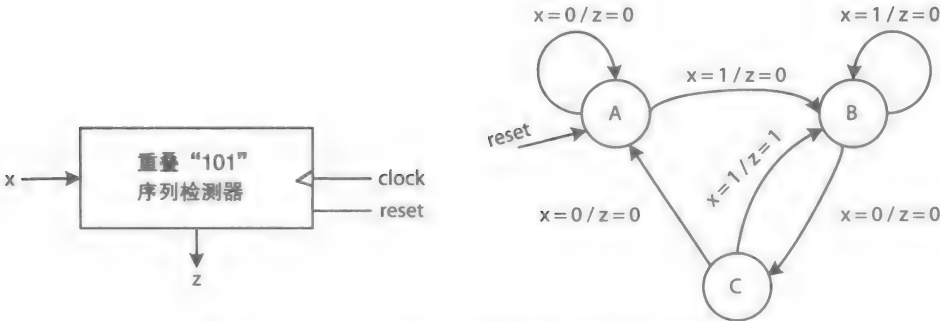


图 5-20 “101” 序列识别器框图及其 Mealy FSD

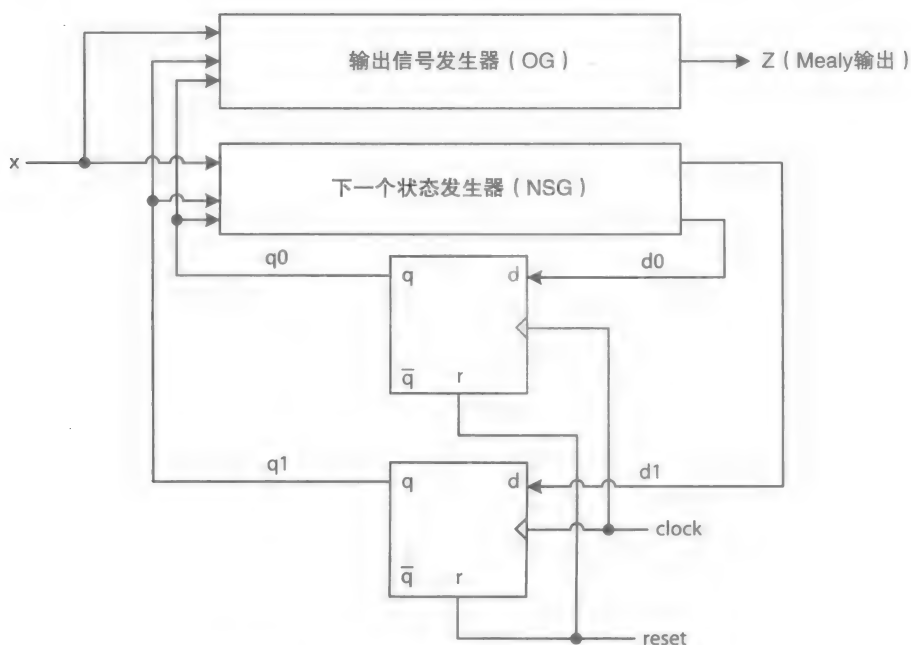
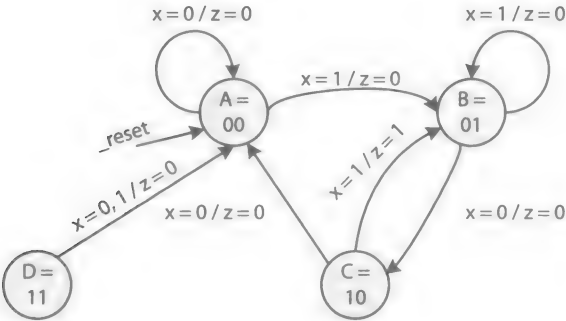


图 5-21 “101” Mealy 序列识别器的详细框图

- **在设计时将 FSM 中未知的状态忽略掉**——在这种情况下，未知的状态被视为无关紧要，例如，假设 FSM 正在操纵一个简单的玩具，如果 FSM 进入一个未知的状态中导致了失灵，这个 FSM 就需要重启。更重要的是，这些未知状态的二进制标识被列入了 NSG 和 OG 的真值表中，但是下一个状态的值和输出变量被设为不关心。这样会减少 NSG 和 OG 电路的尺寸。
- **FSM 未知的状态转换到已知的状态**——在这种情况下，每次 FSM 进入一个未知的状态时，机器将会在下一个时钟周期进入一种已知的状态。例如，图 5-22 表示出一个带有未知状态 D 的二进制编码有限状态图。如果由于环境影响造成 FSM 偶然进入未知状态 D。就像图中所示，它将在下一个时钟周期转换到已知的状态 A，而且不会产生无效的输出。在这种情况下，未知状态 D 的标识将被列入 NSG 和 OG 的真值表中，但是状态 A 将作为下一个状态被标识，而不论此时 $x = 0$ 还是 $x = 1$ 。而且，输出 z 将被置为 0。
- **将时序电路设计为容错 FSM**——在这种情况下，FSM 有能力从未知的状态中恢复，或有能力从意外转换到的一个已知状态中恢复过来，进而继续正常工作。例如，一个简单的位错误可以置位或复位一个触发器，这种位错可以被附加的硬件电路检测出来并加以纠正。这种附加硬件电路可以实现一位检错与纠错方案，容错 FSM 可以使用附加的触发器和逻辑电路去检测并修正异常造成的错误。注意，如果使用独热码标识，错误检测会更简单一些。我们将在 5.5 节中讨论容错 FSM。

假设第一个选项（忽略未知状态）是用来设计图 5-20 中的 Mealy 有限状态图，表 5-6 为设计中 NSG 的真值表，而且包含了被忽略的未知状态 D，当当前状态是 D 时，下一个状态被定义为不关心。同样，表 5-7 为 OG 的真值表，而且当当前状态是 D 时，输出 z 被置为不关心。公式 (5-5) 列出了下一个状态变量 d_1 和 d_0 以及输出变量 z 的最小表达式。



未知状态

图 5-22 带有一个未知状态 D 的二进制编码 FSD

$$d_1 = q_0 \bar{x}$$
$$d_0 = x$$
$$z = q_1 x$$

(5-5)

表 5-6 图 5-20 中的 FSD 对应的 NSG 真值表

| 当前状态 | | 输 入 | 下一个状态 | | | |
|------|-------|-------|-------|-------|-------|---|
| | q_1 | q_0 | x | d_1 | d_0 | |
| A | 0 | 0 | 0 | 0 | 0 | A |
| | 0 | 0 | 1 | 0 | 1 | B |
| B | 0 | 1 | 0 | 1 | 0 | C |
| | 0 | 1 | 1 | 0 | 1 | B |
| C | 1 | 0 | 0 | 0 | 0 | A |
| | 1 | 0 | 1 | 0 | 1 | B |
| D | 1 | 1 | 0 | d | d | |
| | 1 | 1 | 1 | d | d | |

表 5-7 图 5-20 中的 FSD 对应的 OG 真值表

| | 当前状态 | | 输 入 | 输 出 |
|---|-------|-------|-----|-----|
| | q_1 | q_0 | x | z |
| A | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 |
| B | 0 | 1 | 0 | 0 |
| | 0 | 1 | 1 | 0 |
| C | 1 | 0 | 0 | 0 |
| | 1 | 0 | 1 | 1 |
| D | 1 | 1 | 0 | d |
| | 1 | 1 | 1 | d |

图 5-23 中给出了相应的 FSM 电路。通常情况下，Mealy 状态机比与它等效的 Moore 状态机需要更少的触发器。然而，有时 Moore 输出是首选。这种情况下，Mealy 每一个输出会
用一个额外的触发器将 Mealy 输出转换成相应的 Moore 输出。在这个图中，一个触发器将
z-Mealy 转换成了等效的 z-Moore。z-Mealy 异步地取决于外部的输入 x，而 z-Moore 不会。

z-Moore 会使 z-Mealy 滞后一个时钟周期。

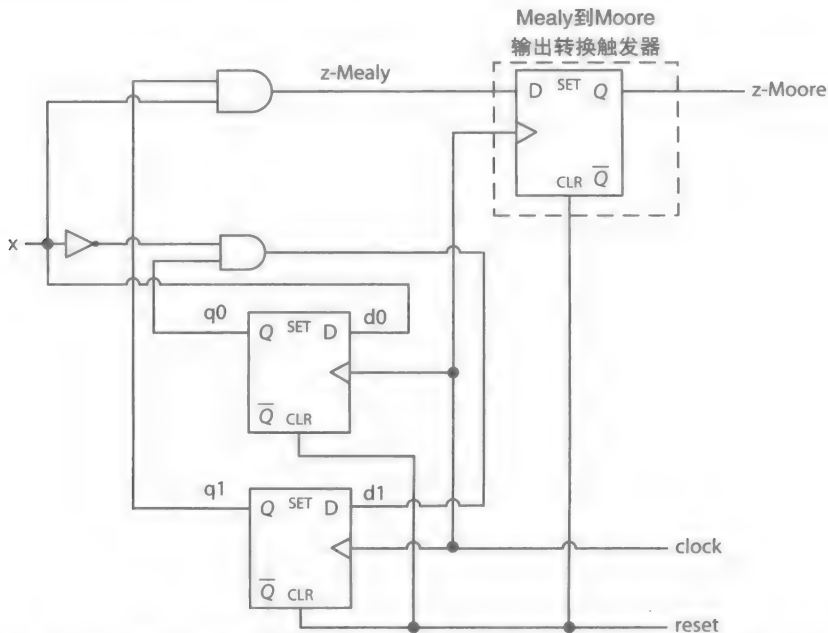


图 5-23 Mealy “101” 序列识别器电路，也表示了用同步触发器实现的 Mealy 输出到 Moore 输出的转换

5.4 计数器

在之前的章节中，介绍过多种 FSM 设计技巧和实例应用。计数器是一个能够输出一组有限指定值的时序电路。例如，一个两位的二进制计数器有序并且重复地输出二进制数 00、01、10 和 11，这个计数器也称为**模 4 计数器**， $(3 + 1) \bmod 4 = 0$ 。通常情况下，一个模 k 的计数器会从 0 开始计数到 $k - 1$ ，并且重复输出。

还有很多其他的计数器的例子。一个**二进制编码的十进制 (BCD)** 的计数器会重复输出序列 0 ~ 9。**格雷码**计数器会输出一系列数字，而且这些数字中每一个数字都与前面的立即数仅有一位不同。例如，数字 000、001、011、010、110、111、101、100，然后后面重复性地又是 000，就可以作为一种 3 位格雷码计数器的输出。

184

格雷码作为一个由完全独立的时序电路产生的外部数据，其两个连续的数据之间仅有一位不同的意义在于降低可能出错的位数为 1 位。示例中表明为什么在用两个独立的功能时序电路去访问先入先出缓冲区时使用格雷码计数器是必要的，请看练习 5.32（也可看 5.6.2 节）。

例 5-3 带有异步高电平有效复位的位串行模 8 计数器的设计使用了 3 个 1 位的计数器片。当 $k = 8$ 时，计数器重复地输出 0 ~ 7。这将涉及同时带有 Mealy 输出和 Moore 输出的混合 FSM 的设计。

解：一个 K 位的二进制计数器片必须完成两种操作之一：保持它的当前记数或者加 1。与最低有效位相关的那一片必须每个时钟周期加 1。其他的片只有在收到指示“加 1”信号的时候才会加 1。图 5-24 为 1 位二进制计数器片的框图和有限状态图。有限状态图中定义了一个带有 Moore 输出 z-Moore 和 Mealy 输出 o-Mealy 的混合 FSM。每一片中的输入信号 i

被用作使能输入信号。如果 $i = 1$ ，这个片会使它的当前值加 1。否则，它将保持当前值。每一个 *o*-Mealy 输出都连到了它迅速响应的片的 i 输入上。

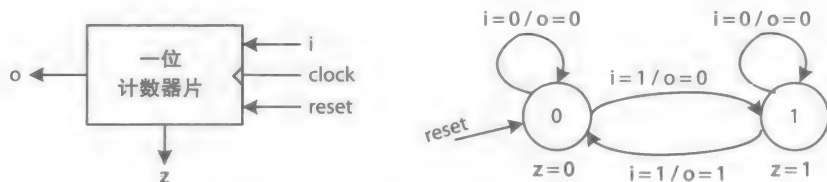


图 5-24 1 位计数器片的框图和有限状态图

表 5-8 为由 1 位计数器片的有限状态图决定的 NSG 和 OG 转换 (真值) 表。公式 (5-6) 中列出了关于 z -Moore 输出、 o -Mealy 输出和下一个状态相关的变量 d 的最小表达式。图 5-25 中给出 1 位计数器片和相应的模 8 计数器的电路。只要 $k \geq 2$, 而且是 2 的幂 (即 $k = 2^m$, 其中 $m \geq 1$), 任何的模 k 计数器都可以由 1 位的计数器片设计而成。

$$\begin{aligned} z &= q \\ o &= q \cdot i \\ d &= q \oplus i \end{aligned} \quad (5-6)$$

表 5-8 图 5-24 中有限状态图决定的 NSG 和 OG 的真值表

| 当前状态 | 输 入 | 下一个状态 | 输 出 | | 备 注 |
|------|-----|-------|---------|---------|-----|
| q | i | d | z-Moore | o-Mealy | |
| 0 | 0 | 0 | 0 | 0 | 保持 |
| 0 | 1 | 1 | 0 | 0 | 增 1 |
| 1 | 0 | 1 | 1 | 0 | 保持 |
| 1 | 1 | 0 | 1 | 1 | 增 1 |

185

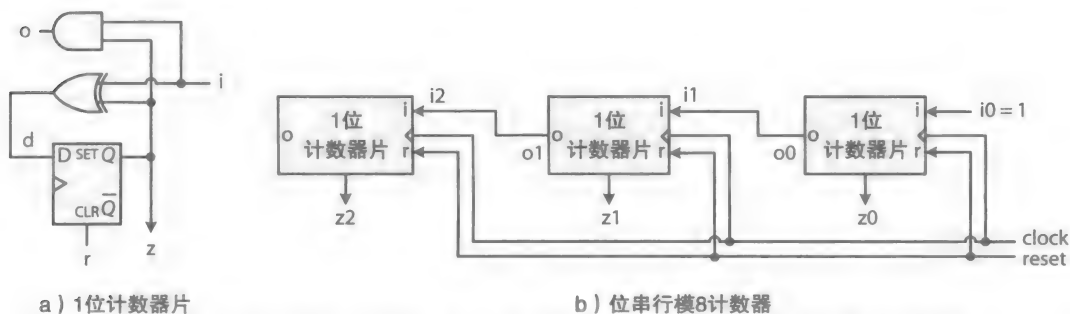


图 5-25 位串行模 8 计数器: a) 1 位计数器片; b) 由 3 片组成的模 8 计数器

例 5-4 带有高电平有效复位信号的位串行模 6 计数器, 注意 6 不是 2 的幂次方。

解：一个模 6 计数器重复输出 0, 1, 2, 3, 4, 5。就像一个模 8 计数器，它也需要 3 个触发器存储一个 0 ~ 5 的数值来作为它的当前状态。然而，与模 8 计数器有所不同，当模 6 计数器计数到 5 时需要重新开始，在下一个时钟周期从 0 开始计数。为了达到这样的目的，一个简单但不首选的解决方案是一旦计数器计数到 6，在下一个时钟到来之前用一个简单的组合电路的输出去复位计数器。一个简单的被标记为 `aclear`（“a”表示异步）的信号被定义为 $aclear = z_1 z_2 \bar{z}_0$ ，如图 5-26 所示，再加入主设备异步信号 `reset` 异步地复位每一片。

187 以随时改变。

解：图 5-29 所示为模 4 上 / 下计数器片的框图和有限状态图。这种片在向上计数时重复输出序列 0, 1, 2, 3, 在向下计数时重复输出序列 3, 2, 1, 0。信号 u 代表计数器的方向。如果 $u = 1$, 计数器向上计数, 如果 $u = 0$, 计数器向下计数。对于一个模 16 的计数器, 第一片负责至少两个重要的位, 每个时钟周期计数一次, 第二片每 4 个时钟周期才计数一次。如果插入输入信号 i , 在下个时钟周期内会使能相关的片。禁用第一片将会自动地禁用第二片。然而当第一片向上计数达到了它的最大值 $value = 3$ 时, 或向下计数达到了它的最小值 $value = 0$ 时, 第二片才会被使能。例如, 在向上计数时当前的计数值 $count = (0011)_2$, 两个片都必须被使能产生下个计数值 $count = (0100)_2$ 。也就是说, 第一片必须向上计数产生 $0 = (00)_2, (3 + 1) \bmod 4 = 0$, 第二片必须向上计数产生 $1 = (01)_2, (0 + 1) \bmod 4 = 1$ 。

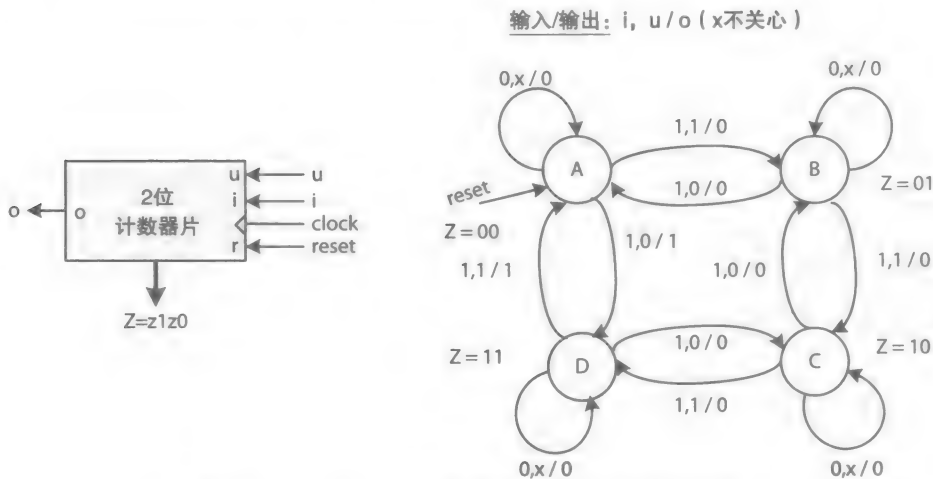


图 5-29 2 位上 / 下计数器片的框图和有限状态图

这个设计需要两个触发器, 因此会有两个当前状态变量 q_1 和 q_0 , 还有两个下一状态变量 d_1 和 d_0 。表 5-10 为 NSG 和 OG 的真值表。然后列出了相关的 Espresso 最小化 SOP 逻辑条件。图 5-30 是模 16 计数器使用两个计数器片的最终电路, 两个计数器片分别被标记为片 1 和片 0。片 0 表现为一直使能 (即 $i_0 = 1$), 而片 1 仅在向上计数片 0 输出为 3 或向下计数片 0 输出为 0 时才被使能。

188

表 5-10 图 5-29 中 2 位计数器片的 NSG 和 OG 的真值表

| | 当前状态 | | 输 入 | | 下一个状态 | | 输 出 | | |
|---|-------|-------|-----|-----|-------|-------|-------|-------|-------|
| | | | | | | | Moore | | Mealy |
| | q_1 | q_0 | i | u | d_1 | d_0 | z_1 | z_0 | o |
| A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| B | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

(续)

| | 当前状态 | | 输 入 | | 下一个状态 | | 输 出 | | |
|---|-------|-------|-----|-----|-------|-------|-------|-------|-------|
| | | | | | | | Moore | | Mealy |
| | q_1 | q_0 | i | u | d_1 | d_0 | z_1 | z_0 | o |
| C | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| D | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |

189

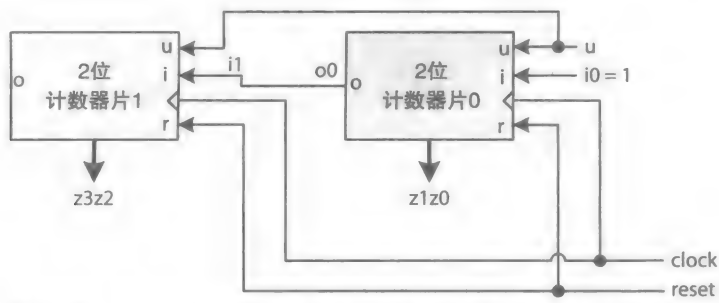


图 5-30 由 2 位上 / 下计数器片实现的位串行模 16 上 / 下计数器

```
#mod-4 counter slice NSG and OG modules
#Input signal labels
#output bit label
.i 4
.o 5
.ilb q1 q0 i u
.ob d1 d0 z1 z0 o
.p 10
0111 10000
0010 10001
1111 00101
10-1 10000
-10- 01000
-01- 01000
11-0 10100
10-- 00100
1-0- 10100
-1- 00010
.e
```

另外，因为所有的计数器都需要某种形式的加法器来表现出一个已知的功能，所以它们的设计可以不需要有限状态图。这将在下面的实例中体现。

例 5-6 一个位并行的模 8 向上计数器，其设计中用到了一系列已知的组合电路（CC）模块和一个并行加载寄存器。

解：图 5-31 为一个位并行模 8 向上计数器的数据通路。这个数据通路包括了一个 3 位的二进制加法器，一个 3 位的 2-1 选择器，还有一个带有异步高电平有效复位信号的 3 位并

行加载寄存器。这个寄存器也是由带有使能端的触发器设计而成的。加法器始终输出寄存器的值加 1。信号 sc (同步清零) 控制着选择器, 用以同步地初始化计数器为 0。在每个时钟周期内, 如果寄存器被使能, 将会加载 3 位的选择器的输出, 即 $sc = 0$ 时的 $Z + 1$ 的值或者 $sc = 1$ 时的 0。

190

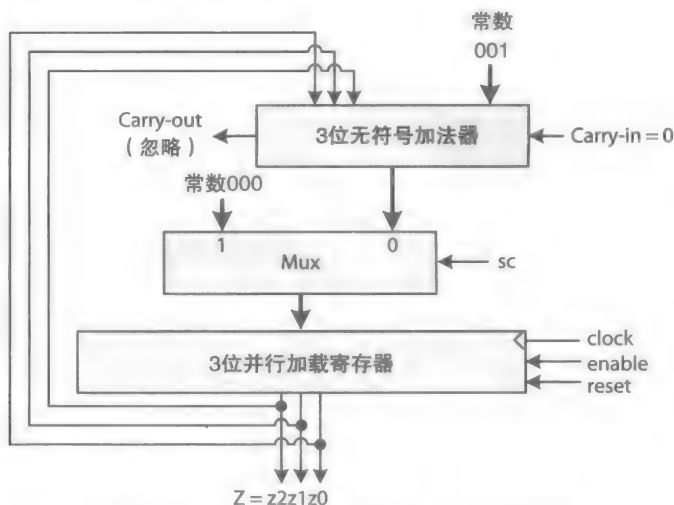


图 5-31 同步清零位并行模 8 向上计数器

计数器的速度取决于加法器的传播时延。例如, 进位传递加法器 (CPA) 会产生较长的信号时延, 和位串行电路比较相似。另一方面, 也可以使用一个更快的加法器, 比如使用超前进位 (CLA) 加法器去设计一个高速的计数器。图 5-32 是由一个 3 位的 CLA 加法器设计实现的带有同步清零的模 8 向上计数器的简化电路, 是使用 CLA 加法器的常数操作数 $(001)_2$ 和选择器的常数输入 $(000)_2$ 情况下的 CLA 和选择器逻辑表达式的进一步简化电路。

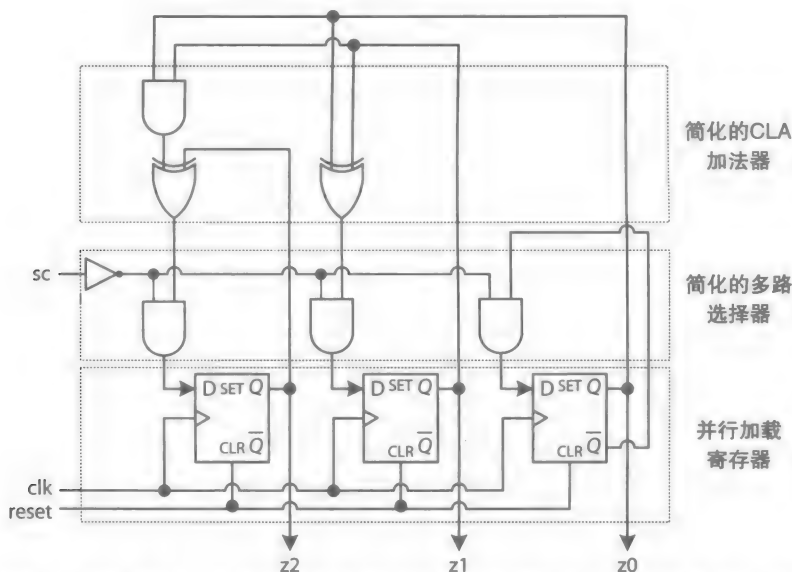


图 5-32 使用简化 CLA 加法器和简化选择器的同步清零位并行模 8 向上计数器

191

为了设计出一个 k 位的位并行计数器片，这个片必须包含 i 和 o 接口信号，而且这应在位串行计数器设计之前进行讨论。另外，位并行（包括位并行片）方案，尤其是在大型设计中，有一个优点就是不需要有限状态图。然而，这种技术需要设计师有能力从设计问题的描述去确定组合电路将会表现出的功能。在第 8 章中，我们将会通过这种方式设计一个 CPU 数据通路。

5.5 容错 FSM

容错 FSM 能够检测并纠正出现的错误，这种错误的出现不是因为工业上或设计上的错误，而是工作过程中随机的环境因素。通常情况下，这些因素会影响存储单元，比如锁存器、触发器和内存。由 q 位表示的触发器的状态可以突然间由 0 变为 1 或由 1 变为 0。一个错误可以使 FSM 转换到一种无效的状态，进而导致电路故障。一个错误还可以导致计数器突然输出一个错误的值，致使序列识别器检测一个错误的序列，或者跳过而未能检测到一个正确的序列等。

通常情况下，错误会影响到一位或多位。然而，一位的错误更常见。容错 FSM 需要额外的硬件去实现冗余，能够检测并纠正错误引起的故障。例如，容错 FSM 需要额外的触发器和额外的组合电路。额外触发器的数量取决于最初的有限状态图中有多少状态。例如，设计一个一位容错 FSM，每一个二进制状态标识与其他的标识相比至少有 3 位不同。通常情况下，两个二进制数间不同的位数叫作汉明间距。而且，如果一系列数中每一个数都与其他数有至少三个汉明间距，那么这些数被称为汉明码。

例如，如果一个有限状态图有三种状态，那么一个一位容错 FSM 必须使用 5 位的汉明码，如 00000、00111 和 11001 来标识这三种状态。注意，任何三个 5 位的汉明码的汉明间距是 3 或者更多。码 00000 和码 00111 的前三位不同，因此它们的汉明间距是 3。码 00111 和码 11001 的不同位是 1、2、3 和 4（小端），因此它们的汉明间距是 4。任何两个码的汉明间距是它们的位异或结果中 1 的个数。表 5-11 表示了三个码 00000、00111 和 11001 计算后的汉明间距。

表 5-11 每对码的汉明间距

| Code 1 | Code 2 | Code 1 \oplus Code 2 | 汉明间距 |
|--------|--------|------------------------|------|
| 00000 | 00111 | 00111 | 3 |
| 00000 | 11001 | 11001 | 3 |
| 00111 | 11001 | 11110 | 4 |

假设一个错误导致码 00000 的位 2 由 0 变为了 1，产生了一个无效的码 00100。这个新的码与有效的 00000 之间的汉明间距是 1，但是与有效的码 00111 和 11001 中每一个的汉明间距是 2 或者更多。因此，这个无效的码与有效的码 00000 之间的距离比其他两个有效码之间的距离更小一些。因此，这个由错误导致的异常可以被检测出来，并能加以纠正，即将无效的码 00100 由有效的码 00000 替换掉。

例 5-7 图 5-33 中给出了由 Mealy 有限状态图得到的容错 FSM 的设计。如图中所示，每个状态都由一个 5 位的汉明码标识。

解：表 5-12 为使用 5 个触发器实现的容错 FSM 的 NSG 和 OG 真值表。在这个表中，所有一位无效的当前状态标识和相应有效的状态标识是一样的。例如，因为每一个无效的标识

00001、00010、00100、01000 和 10000 与有效的码 00000 是 1 个汉明间距，所以它们都被视为状态 A。在这个表中的前 6 栏中，如果 $x = 0$ ，那么下个状态是 00000 (状态 A)。因此，信号 q_4 、 q_3 、 q_2 、 q_1 和 q_0 中任意的 1 位变化都不会改变 FSM 的状态。最终的电路没有展示出来。然而，可以通过使用 Espresso 最小化软件来简化真值表，获得电路所需的逻辑表达式。另外，电路可以在 Verilog 中通过使用 case 语句进入真值表来给电路建模 (见练习 5.25)。

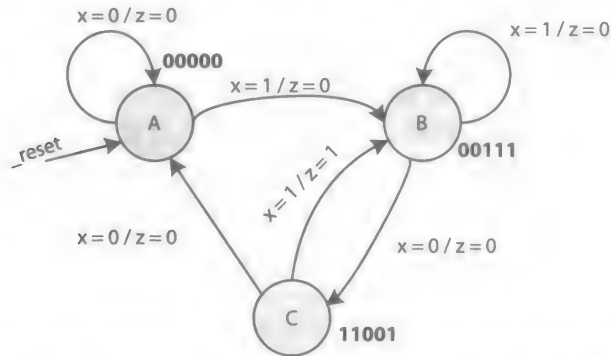


图 5-33 使用 5 位汉明状态码作为状态标识的有限状态图

表 5-12 图 5-33 中有限状态图决定的容错的 NSG 和 OG 真值表

| 当前状态 | | | | | | 输入 | 下一个状态 | | | | | 输出 |
|-------|-------|-------|-------|-------|---|-----|-------|-------|-------|-------|-------|-----|
| q_4 | q_3 | q_2 | q_1 | q_0 | | x | d_4 | d_3 | d_2 | d_1 | d_0 | z |
| A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | A |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| B | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | B |
| | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| C | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | C |
| | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | |
| | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | |
| | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | |
| | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | |
| | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | B |
| | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |

(续)

| 当前状态 | | | | | 输 入 | 下一个状态 | | | | | 输 出 |
|-------|-------|-------|-------|-------|-----|-------|-------|-------|-------|-------|-----|
| q_4 | q_3 | q_2 | q_1 | q_0 | x | d_4 | d_3 | d_2 | d_1 | d_0 | z |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | A |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| C | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | B |
| | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | |
| | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

如果通过单独控制的输入信号来复位或预置每一个触发器，那么就可以仿真一个简单的错误。例如，假设当前的状态是 00000，通过使用相关的预置信号改变状态为 00100（一个无效的状态），那么我说，一个错误被引入位 q_2 中。正如表 5-12 中所示，FSM 应当在 $x = 0$ 时由无效状态正确地转换到有效状态 00000 中，或在 $x = 1$ 时由无效状态正确地转换到有效状态 00111 中。因此，这个电路将能检测并纠正一位错误。注意在这种情况下，检测和纠正机制将被嵌入在 NSG 和 OG 模块中。接下来将会讨论另外一种技术，这种技术将会把非容错处理的时序电路转换为一位容错的电路，并且，这种技术不需要建立大型的真值表。

另外，通常而言，很容易想出一个小的汉明码数字去设计实现一个小的容错 FSM。然而，使用即将介绍的汉明编码方案为任意规模有限状态图生成尽可能多的必要汉明码会更为合理。

汉明编码方案

汉明码可以用来检测并纠正一位错误或检测两位错误。这些错误可能发生在数字通信的数字传输或者在数据存储中，比如触发器或内存中。每个汉明码中包括一定数量的奇偶校验位和一定数量的数据位。奇偶校验位可以被计算为奇校验或偶校验。例如一个 7 位的汉明码，这些位从右到左被编号为 1 ~ 7，其中，位 1、2 和 4 被保留为三位偶校验位，而位 3、5、6 和 7 被作为 4 位数据位。公式 (5-8) 用来计算从数据位 d_3 、 d_5 、 d_6 和 d_7 得到的偶校验位 p_1 、 p_2 和 p_4 。现在假设 $d_7 = 1$ ， $d_5 = 1$ 和 $d_3 = 0$ 。校验位 p_1 作为一个偶校验位必须是 0，这样 d_7 、 d_5 、 d_3 和 p_1 中 1 的个数才是偶数。如果 $d_7 = 1$ ， $d_5 = 1$ 和 $d_3 = 1$ ，那么 p_1 就得是 1。

$$p_1 = d_3 \oplus d_5 \oplus d_7$$
$$p_2 = d_3 \oplus d_6 \oplus d_7$$
$$p_4 = d_5 \oplus d_6 \oplus d_7$$

(5-8)

给出一个 4 位的数 1101，它的 7 位汉明码由公式 (5-8) 决定，如下所示：

$$\begin{aligned}
 \text{数据位: } d_3 &= 1, d_5 = 0, d_6 = 1, d_7 = 1 \\
 \text{偶校验位: } p_1 &= 1 \oplus 0 \oplus 1 = 0 \\
 p_2 &= 1 \oplus 1 \oplus 1 = 1 \\
 p_4 &= 0 \oplus 1 \oplus 1 = 0
 \end{aligned}$$

这个 7 位的汉明码被组织为 $d_7d_6d_5p_4d_3p_2p_1 = 1100110$ 。校验位 p_1 由数据位 d_3 、 d_5 和 d_7 决定。这些数据位分别在汉明码的第 $01\bar{1}$ (3) 位、第 $10\bar{1}$ (5) 位和第 $11\bar{1}$ (7) 位。注意每个位置数字的第一位都是 1, 已由下划线表示了出来。同样, 决定校验位 p_2 的数据位于位置 $01\bar{1}$ (3)、 $11\bar{0}$ (6) 和 $11\bar{1}$ (7) 上。通常而言, $k = 0, 1, 2, \dots$ 时, 相应校验位 p_2^k 分别由位数所在位置数中第 k 位都为 1 的那些位数进行异或得到。例如, $k = 2$, p_4 由异或所有位置为 5 (101)、6 (110) 和 7 (111) 上的位数所得, 因为 5 是 $4 + 1$, 6 是 $4 + 2$, 7 是 $4 + 3$, 4 是它们的共有值。

校验位用来确定错误位的位置 (如果有的话)。例如一个 7 位的汉明码 $d_7d_6d_5p_4d_3p_2p_1 = 1001100$, 假设这个汉明码被无线传输到一个很远的目的地。假如收到的汉明码被表示为 $d_7'd_6'd_5'p_4'd_3'p_2'p_1' = (1101100)_2$, 即位 d_6 (有下划线) 上有一个错误。接收的校验位和数据位分别是 $p_4'p_2'p_1' = 100$ 和 $d_7'd_6'd_5'd_3' = 1101$ 。通过公式 (5-8) 可以得到新的校验位, 校验位 p_4'' 、 p_2'' 和 p_1'' 由接收到的数据位 $d_7'd_6'd_5'd_3' = 1101$ 计算可得:

$$\begin{aligned}
 p_1'' &= d_3' \oplus d_5' \oplus d_7' = 1 \oplus 0 \oplus 1 = 0 \\
 p_2'' &= d_3' \oplus d_6' \oplus d_7' = 1 \oplus 1 \oplus 1 = 1 \\
 p_4'' &= d_5' \oplus d_6' \oplus d_7' = 0 \oplus 1 \oplus 1 = 0
 \end{aligned}$$

式 (5-9) 用来确定出错位的位置, 表现为一个 3 位的数值 $E = e_2e_1e_0$ 。

$$E = e_2e_1e_0 = p_4''p_2''p_1'' \oplus p_4'p_2'p_1' \quad (5-9)$$

也就是,

$$\begin{aligned}
 E &= e_2e_1e_0 = 010 \oplus 100 \\
 &= 110 \\
 &= 6(\text{出错位的位置})
 \end{aligned}$$

6 表示接收到的汉明码在位数 6 上有错误发生, 应当把这位中的 1 改为 0, 纠正后的汉明码 1001100 才是所发送的。

通过另外一个整体校验位 c , 可以检测到汉明码的双位错误, 而不是纠正。公式 (5-10) 定义了一个整体偶校验位。4 位校验位 c 、 p_4 、 p_2 、 p_1 和 4 位数据位 d_7 、 d_6 、 d_5 、 d_3 组成了一个 8 位的汉明码。表 5-13 给出了一些汉明码位数的例子。

$$c = d_7 \oplus d_6 \oplus d_5 \oplus p_4 \oplus d_3 \oplus p_2 \oplus p_1 \quad (5-10)$$

表 5-13 汉明编码位数举例

| 校验位的位数 | 最大的汉明编码位数 |
|---|----------------------|
| 4 位, p_1 、 p_2 、 p_4 和 c | 8 位, 其中最多 4 位数据位 |
| 5 位, p_1 、 p_2 、 p_4 、 p_8 和 c | 16 位, 其中最多 11 位数据位 |
| 6 位, p_1 、 p_2 、 p_4 、 p_8 、 p_{16} 和 c | 32 位, 其中最多 26 位数据位 |
| 7 位, p_1 、 p_2 、 p_4 、 p_8 、 p_{16} 、 p_{32} 和 c | 64 位, 其中最多 57 位数据位 |
| 8 位, p_1 、 p_2 、 p_4 、 p_8 、 p_{16} 、 p_{32} 、 p_{64} 和 c | 128 位, 其中最多 120 位数据位 |

表 5-14 给出了使用 E 、 c' （接收到的整体校验位）和 c'' （从收到的数据中计算而得的整体校验位）的汉明编码方案（HCS）的规则。当 c' 和 c'' 相等，而且 $E = 0$ 时，表示接收到的汉明码中没有错误。如果 c' 和 c'' 相等，但 $E \neq 0$ ，表示在收到数据中有两位不可纠正的错误。如果 $c' \neq c''$ 且 $E = 0$ ，就表示 c' 发生了错误。最后，如果 $c' \neq c''$ ，而且 $E \neq 0$ ，那么表示 E 发生了错误。如果 3 位或 3 位以上发生了错误，HCS 将会把错误（错误性地）解释为一个单位错误或一个双位错误。

表 5-14 HCS（汉明编码方案）的规则

| 接收到的整体校验位进行运算 vs 由接收到的数据位 $c' \oplus c''$ 所得 | 出错指示位 | 含 义 |
|--|------------|--------------------|
| 0 | $E = 0$ | 没有错误 |
| 0 | $E \neq 0$ | 一个两位不可纠正的错误 |
| 1 | $E = 0$ | 接收到的整体校验位 c' 出错 |
| 1 | $E \neq 0$ | 位置 E 上出现一位可纠正的错误 |

例 5-7 中，我们选择了三个汉明码 00000、00111 和 11001 来标识图 5-33 中有限状态图的三种状态。这里，我们将会解释 HCS 是如何使用汉明码系统地对有限状态图的状态进行编码的。假设图 5-33 中的状态 A、B 和 C 被 2 位的二进制数进行了初始化编码，其中 $s_1s_0 = 00$ 为状态 A，01 为状态 B，10 为状态 C。在用两位数据位作为 2 位状态标识时，公式（5-11）使用数据位 $d_7 = 0$ ， $d_6 = 0$ ， $d_5 = s_1$ 和 $d_3 = s_0$ 来计算校验位。

$$\begin{aligned} p_1 &= s_0 \oplus s_1 \\ p_2 &= s_0 \\ p_4 &= s_1 \end{aligned}$$

(5-11)

例如，当 $s_1s_0 = 01$ 时， $p_1 = 1$ ， $p_2 = 1$ ， $p_4 = 0$ ，对应的汉明码是 $s_1p_4s_0p_2p_1 = (00111)_2$ 。表 5-15 给出了校验位计算的一个总结。可以看到，计算出的汉明码和例 5-7 中所使用的一样。

表 5-15 由 2 位数字 00、01 和 10 生成的 5 位汉明码

| 状态 | 初始的二进制状态标识 | | 校验位 | | | 5 位汉明码 |
|----|------------|-------|-------|-------|-------|-------------------|
| | s_1 | s_0 | p_1 | p_2 | p_4 | $s_1p_4s_0p_2p_1$ |
| A | 0 | 0 | 0 | 0 | 0 | 00000 |
| B | 0 | 1 | 1 | 1 | 0 | 00111 |
| C | 1 | 0 | 1 | 0 | 1 | 11001 |

对于例 5-7 中讨论的容错 FSM，它首先包含一个作为非容错电路的 FSM，然后在电路中又合并了一个额外的电路，用于实现汉明码检错与纠错机制。例如，在例 5-7 中的容错 FSM 设计问题中，首先，FSM 被设计为一个非容错 FSM。这将需要两个触发器（两个状态位）、一个 NSG 和一个 OG。假设由 NSG 产生的两个下一状态位被标记为 s_1 和 s_0 。一个容错的电路将需要 6 个触发器，分别作为两个状态位 s_1 和 s_0 ，三个校验位 p_1 、 p_2 和 p_4 ，还有一个整体校验位 c 。

假设这 6 个触发器的 q 位分别被表示为 $q_0 \sim q_5$ 。对于一个容错的设计，这些触发器可以被解释为一个发送介质和一个接收器。这些 d 位通过触发器被“发送”出去，然后作为 q 位被“接收”到。简单地反转一个 q 位便可能造成错误。在发送端，这 6 个状态位（ d 's）将被连接到信号 s_1 、 s_0 和 4 个校验位 p_1 、 p_2 、 p_4 和 c 上。在接收端，由 q 表示两个当前状态位

196

197

s'_1 、 s'_0 和 4 个接收到的校验位 p'_1 、 p'_2 、 p'_4 和 c' 。表 5-16 总结了汉明码和触发器输入输出的关系。

表 5-16 使用汉明码检错与纠错机制将 FSM 转换为容错 FSM

| 触发器的 d 位 (“发送的”) | 触发器的 q 位 (“接收的”) | 由接收的状态位计算所得的校验位 | 错误位置 |
|--------------------|--------------------|---|---|
| $d_0 = p_1$ | $p'_1 = q_0$ | $p''_1 = q_2 \oplus q_4$ | $E = p''_4 p''_2 p''_1 \oplus p'_4 p'_2 p'_1$ |
| $d_1 = p_2$ | $p'_2 = q_1$ | $p''_2 = q_2$ | |
| $d_2 = s_0$ | $s'_0 = q_2$ | $p''_4 = q_4$ | |
| $d_3 = p_4$ | $p'_4 = q_3$ | $c'' = q_4 \oplus q_3 \oplus q_2 \oplus q_1 \oplus q_0$ | |
| $d_4 = s_1$ | $s'_1 = q_4$ | | |
| $d_5 = c$ | $c' = q_5$ | | |

一个明显的容错 FSM 的设计需要两个额外的模块：奇偶发生器 (PG) 模块和检错与纠错 (EDC) 模块。PG 模块将会输入 s_1 和 s_0 ，然后产生 4 个偶校验位 p_1 、 p_2 、 p_4 和 c 。EDC 将会输入位 p'_1 和 s'_0 (“接收”的下一状态位)，然后产生校验位 p''_4 、 p''_2 和 p''_0 。如果 $c' \neq c''$ ，位 p'_4 、 p'_2 和 p'_0 以及 p''_4 、 p''_2 和 p''_0 将被用来计算错误位置 E (公式 (5-9))。

如果 $E \neq 0$ ，使用 3-8 译码器， E 将被译码为 7 个输出信号 (1 ~ 7) 中的一个，而且是仅有的一个有效信号。如果 $c' \neq c''$ ，这个有效的编码信号 (如果有) 将随后通过异或门被用来纠正错误位。然而，这种合成的 FSM 将比由例 5-7 中真值表和 SOP 或 POS 表达式设计实现的 FSM 的传播延时更长一些。

5.6 时序电路的时序

触发器共享同一个时钟信号，这样它们在采样时钟边沿接收时大约在同一个时刻，以保证在下一个采样边沿到来之前所有的触发器都能同时对它们各自的输入进行采样。否则，如果时钟信号在向触发器的传播过程中有延时存在，那么一些触发器比其他的触发器接收到的采样时钟晚一些的情况是可能存在的。例如图 5-34 中所示，clk1 先到达，一些延时 (由于信号线路延时) 之后 clk2 到达第二个触发器 (FF2)。采样时钟边缘到达不同触发器的时间差异称为时钟脉冲相位差 (τ_{cs})。

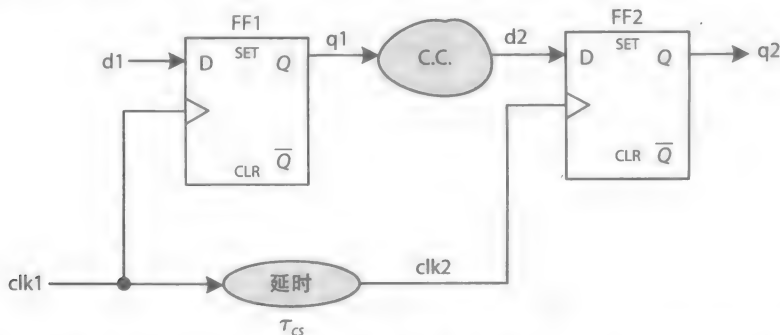


图 5-34 描述一个时钟内可能存在的时钟相位差问题。如果到达 FF2 的采样时钟边缘较晚，信号 d_2 可能会发生变化

时钟相位差会导致很多问题出现。更早一些收到采样时钟边缘的触发器对输入端进行采样，会比其他触发器更早地改变它们的输出。最终，有可能出现最新采样的输入会部分改变

或全部改变那些还未完成输入采样的触发器的输入端值。反过来，这可能会导致时序（建立时间或保持时间）冲突或造成多功能电路中无效的状态转换。

在这个图中，当FF1收到 clk1 的采样边沿时，它开始对 $d_1^{current}$ 进行采样，然后将 $q_1^{current}$ 改为 q_1^{new} 。当FF2收到 clk2 的采样边沿时，它应当采样 $d_2^{current}$ ，然后改变 $q_2^{current}$ 为 q_2^{new} 。然而，由于时钟相位差，三种情况之一有可能会发生，见图 5-35。

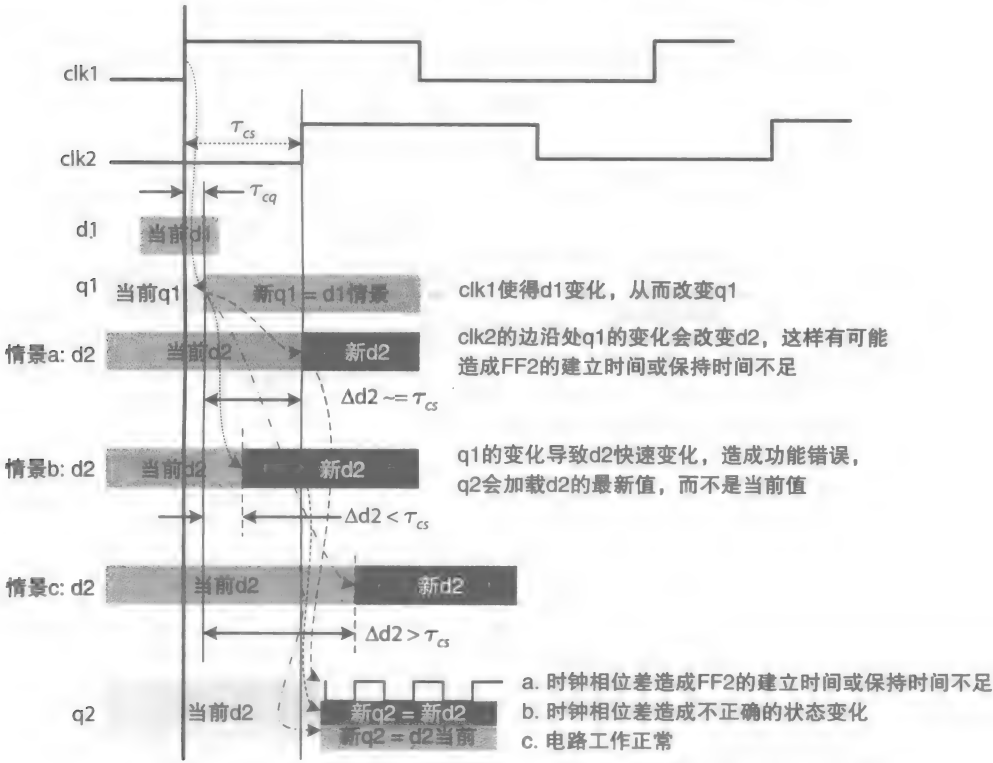


图 5-35 一个时钟内由时钟相位差造成影响的时序图（图 5-34 中电路）

199

情景 a： d_2 的传播时延与时钟相位差是相等的 ($\Delta d_2 \cong \tau_{cs}$)。在这种情况下，当 FF2 仍在采样时， d_2 有可能还在变化，因此有可能造成 FF2 的建立时间或保持时间不足。

情景 b： d_2 的传播时延比时钟相位差小 ($\Delta d_2 < \tau_{cs}$)。在这种情况下， q_1^{new} 将会在 FF2 的 clk2 采样边沿到来之前将 $d_2^{current}$ 改为 d_2^{new} ，这样将会导致在 FF1 加载 $d_1^{current}$ 时，FF2 加载 d_2^{new} 。这将导致无效的状态转换，造成功能错误。

情景 c： d_2 的传播时延比时钟相位差大 ($\Delta d_2 > \tau_{cs}$)。在这种情况下，clk2 的采样边沿会在 q_1^{new} 将 $d_2^{current}$ 改为 d_2^{new} 之前到达 FF2，因此，FF2 将会正常地加载 $d_2^{current}$ ，电路正常工作。

图 5-34 的电路仅包含两个触发器。通常情况下，一个电路可能包含多个触发器，每个触发器的 d 输入端可能取决于一个或多个 q 。在这种情况下，为了让电路正常工作， q^{new} 可以改变 $d^{current}$ 的最早时间是最小的 clock 到 q 的时间 (τ_{cq-min}) 和最小的电路传播时延 (τ_{pd-min}) 之和。式 (5-12) 给出了保持电路正确操作的必要条件：

$$\tau_{cs} < \tau_{cq-min} + \tau_{pd-min} \tag{5-12}$$

否则， $d^{current}$ 将会改变太快，比如图 5-35 中的情景 a 和情景 b，这将会导致建立时间或

保持时间不足 (情景 a) 或造成功能错误 (情景 b)。 τ_{cs} 与 $\tau_{cq-\min} + \tau_{pd-\min}$ 的距离大小可以被定义为 τ_{ht} (保持时间), 其关系在式 (5-13) 中表示了出来:

$$\tau_{cq-\min} + \tau_{pd-\min} - \tau_{cs} \geq \tau_{ht} \quad (5-13)$$

图 5-36 是另外一个可能带有时钟相位差的电路。在这种情况下, clk1 的采样边沿会在 clk2 的采样边沿到达 FF1 之前到达 FF2。

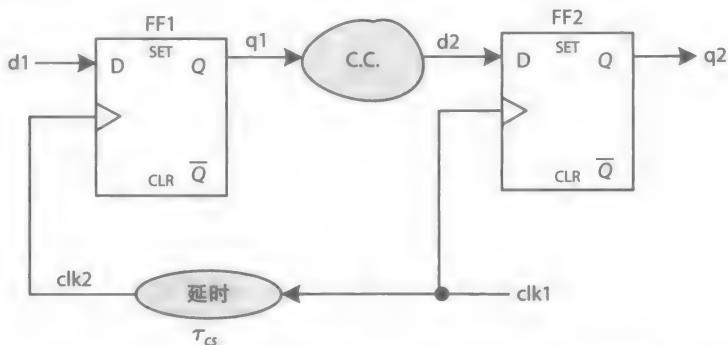


图 5-36 clk1 的当前采样边沿和下一个采样边沿之间可能存在的时钟相位差问题。当 clk1 的下一个采样边沿到达 FF2 时, 信号 d_2 可能正在变化中

在这个图中, 在 clk2 的采样边沿到达 FF1 的时间和下一个 clk1 的采样边沿到达 FF2 的时间之间, 时钟相位差可能会造成两种可能的情景。这两种情景将在图 5-37 中介绍并加以描述:

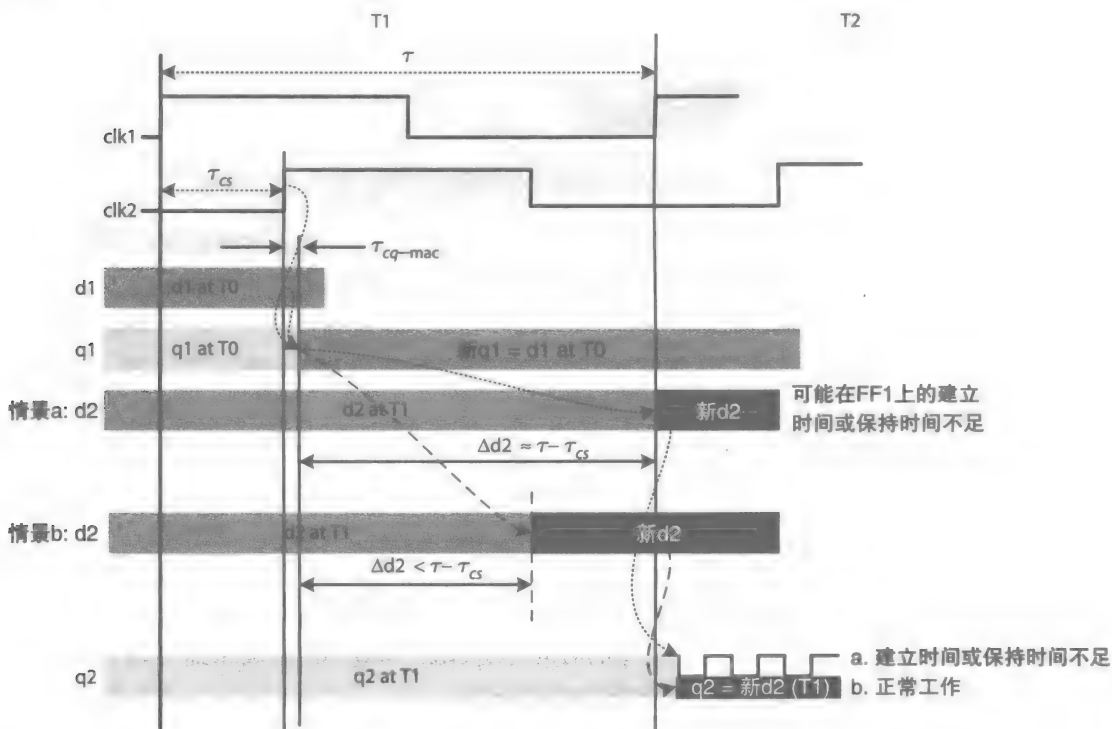


图 5-37 clk1 的当前和下一个采样边沿之间相位差影响的时序图 (图 5-36 电路)

情景 a： d_2 的传播时延和时钟周期 (τ) 与时钟相位差之间的差值是相等的 (即 $\Delta d_2 \cong \tau - \tau_{cs}$)。在这种情况下，当 clk1 的下个采样边沿到达 FF2 时， d_2^{new} 可能还在变化中。因此， d_2^{new} 可能会造成到达 FF2 的建立时间或保持时间不足。

情景 b： d_2 的传播时延比时钟周期 (τ) 与时钟相位差之间的差值小 (即 $\Delta d_2 < \tau - \tau_{cs}$)。在这种情况下， d_2^{new} 在 clk1 的下个采样边沿到来之前保持稳定。因此，FF2 会如预期的那样加载 d_2^{new} 。所以，这个电路将会正常工作。

201

例 5.8 考虑图 5-34 中的电路。假设 $\tau = 0.65\text{ns}$ ， $\Delta_{cc} = 0.25\text{ns}$ ， $\Delta_{delay} = 0.3\text{ns}$ ， $\tau_{st} = 0.05\text{ns}$ ， $\tau_{cq} = 0.05\text{ns}$ 。画一个时序图并讨论是否存在由时钟相位差导致的问题。

解：时序图如图 5-38 所示。信号 $d_1^{current}$ 在 clk1 的上升沿时被采样，因此在 clk1 边沿之后的 $\tau_{cq} = 0.05\text{ns}$ 处 q_1^{new} 变成了 $d_1^{current}$ 。这时， q_1^{new} 开始改变 d_2 ，从 $q_1^{current}$ 变为 q_1^{new} 开始计时， $d_2^{current}$ 需要 $\Delta_{cc} = 0.25\text{ns}$ 的时间去改变为 d_2^{new} 。这时的时间是 $\Delta_{cc} + \tau_{cq}$ ，或者说 0.3ns 。因此，当 FF2 开始采样 d_2 时， d_2 将会变化，最终导致在 FF2 的建立时间不足。

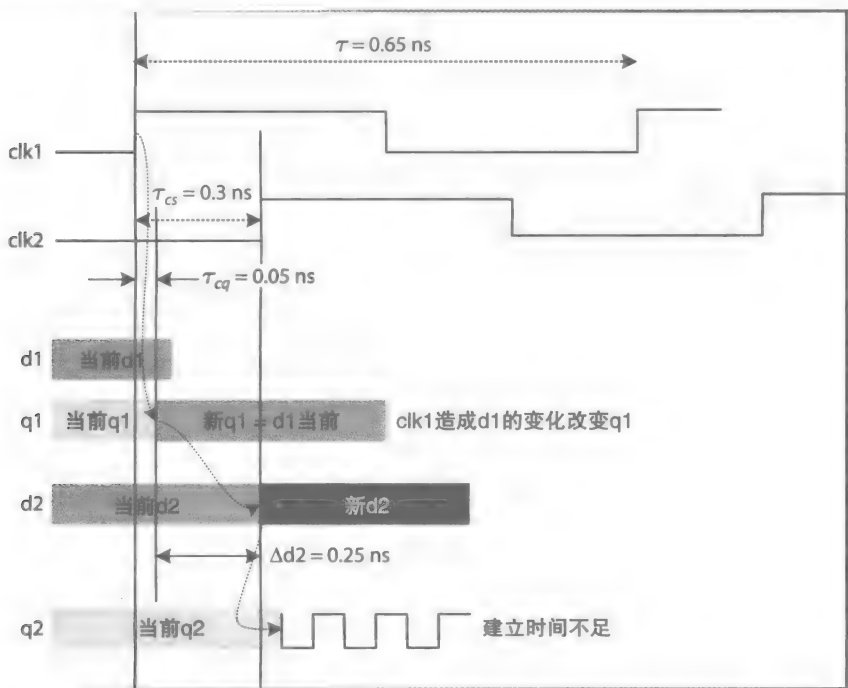


图 5-38 由时钟相位差造成的建立时间或保持时间不足的时序图

5.6.1 带有时钟相位差的时钟频率评估

总的估计的最小时钟周期已在第 4 章中介绍过，但没有考虑由时钟相位差导致的时间损失。如图 5-37 所示，在下个时钟边沿到来之前，信号稳定所需的总时间减少等于一个时钟相位差的长度。这就意味着最小的时钟周期必须包含由时钟相位差导致的延时，如公式 (5-14) [1] 所示。

$$\tau \geq \tau_{cq-max} + \tau_{pd-max} + \tau_{st} + \tau_{cs}$$

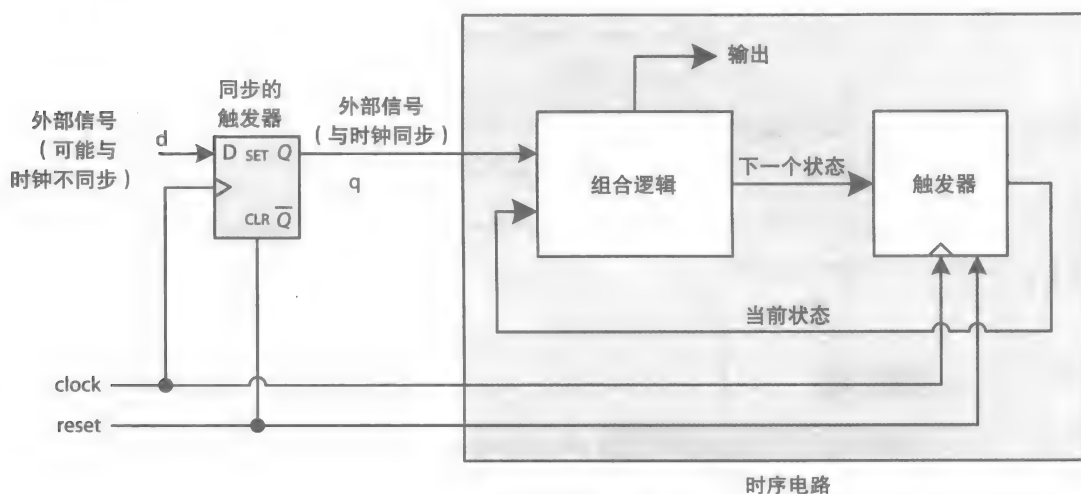
(5-14)

5.6.2 异步接口

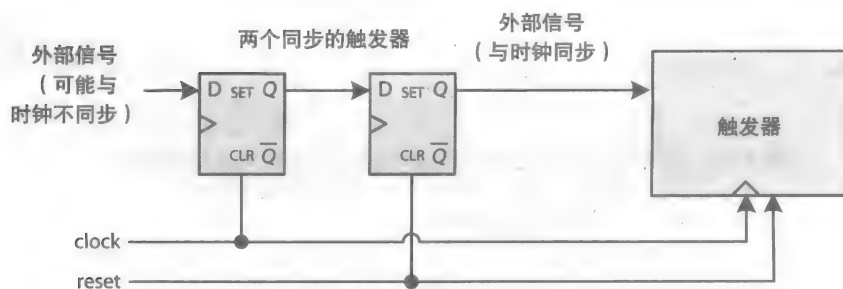
取决于外部输入的时序电路 (如图 5-36 中的 d_1) 期望外部的信号总是在与时钟采样边

沿有关的合适的时间发生变化。然而，如果是由输入设备产生的外部信号可能会随时变化（例如键盘），或者是使用不同时钟源的另一个时序电路的输出，那么最终可能会造成时序电路中触发器的建立时间或保持时间不足，导致亚稳态或可能的故障。

如图 5-39a 所示，一个解决这种问题的建议性方案 [2-3] 是在外部输入进入目标时序电路之前对它们进行采样。在这个图中，外部输入接到同步触发器上。这样在下个时钟边沿到来之前，由输入导致的任何可能的亚稳态都将因为同步触发器而被解决。也就是说，如果输入满足不了同步触发器的建立时间或保持时间而导致触发器输出振荡（亚稳态），预计在下个时钟边沿到来之前，振荡的输出将稳定于 1 或 0。因此，这个时序电路将输入同步的信号，从而可能避免本身的亚稳态性。这些可由图 5-39a 中电路对应的图 5-40 的时序图示例得到说明。而且，可以设计一个同步触发器，这样当外部输入在不合适的时间发生变化造成建立时间或保持时间不足时，它的输出能够快速稳定 [4]。



a) 一个同步的触发器



b) 两个同步的触发器

图 5-39 外部输入同步 [1]: a) 使用一个同步触发器; b) 使用两个触发器允许最大的亚稳态解决时间

然而，为了防止图 5-39a 同步触发器的输出中可能存在的亚稳态进入时序电路中，图 5-39b 中使用了两个同步触发器。

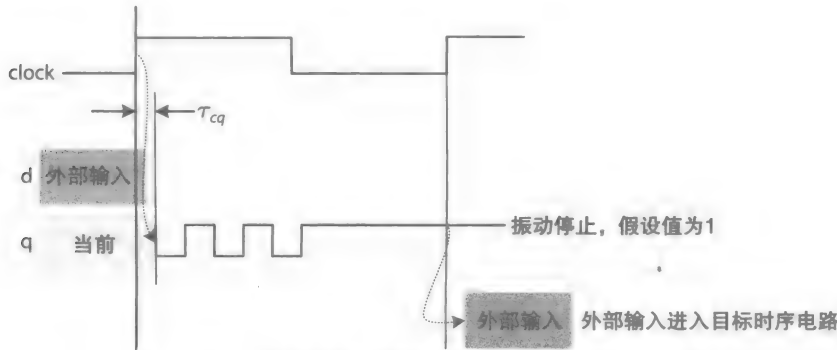


图 5-40 一种解决外部亚稳态输入方案的示例时序图

5.7 硬件描述语言模型

例 5-9 和例 5-10 中介绍了 Mealy 和 Moore FSM 的 HDL 建模。

例 5-9 下面给出例 5-1 中检测重叠序列“101”的 Moore 时序识别器的 Verilog 行为建模。这里的代码被分成如下的三部分：

第 1 部分代码：NSG 的行为描述。它描述了识别器有限状态图的行为，其中有限状态图由被标识为 A ~ D 的 4 个状态组成。

第 2 部分代码：OG 的行为描述。它描述了检测信号为“101”序列的状态，其中识别器输出为 1。

第 3 部分代码：带有异步复位功能触发器的行为描述。reset 有效时，FSM 被初始化为初始状态 A。

HDL 建模

```
//A Moore sequence recognizer that detects the overlapping
//sequence "101".
//Using binary encoded state labels
module moore_seq
(
    input clock, reset, x,
    output reg z
);
//assign binary encoded codes to the states A through D
parameter    A = 2'b00,
              B = 2'b01,
              C = 2'b10,
              D = 2'b11;

reg [1:0] current_state, next_state;
//Section 1: Next state generator (NSG)
always@(*)
begin
    casex (current_state) //ignore unknown and high
                        //impedance (Z) inputs
    A:  if (x == 1)
        next_state = B;
    else
```

```

        next_state = A;
B:  if (x == 1)
        next_state = B;
    else
        next_state = C;
C:  if (x == 1)
        next_state = D;
    else
        next_state = A;
D:  if (x == 1)
        next_state = B;
    else
        next_state = C;
    endcase
end

//Section 2: Output generator (OG)
always@(*)
begin
    if(current_state == D)
        z = 1;
    else
        z = 0;
end

//Sections 3: The flip-flops
always@(posedge clock, posedge reset)
begin
    if (reset == 1)
        current_state <= A;
    else
        current_state <= next_state;
end
endmodule

```

205

仿真测试台

```

`include "moore_seq.v"
module tester();
reg clock, reset, x;
wire z;

moore_seq    u1(clock, reset, x, z);
initial begin
$monitor("%4d: z = %b", $time, z);
clock = 0;
reset = 1;    //reset the flip-flops
x = 0;
#10 reset = 0; //end reset
end
always
begin

```

```

#5clock = ~clock; //generates a clock signal with period 10
end
initial begin //one input per clock cycle
#10 x = 1; $display("%4d: x = %b", $time, x);
#10 x = 1; $display("%4d: x = %b", $time, x);
#10 x = 1; $display("%4d: x = %b", $time, x);
#10 x = 0; $display("%4d: x = %b", $time, x);
#10 x = 1; $display("%4d: x = %b", $time, x);
#10 x = 0; $display("%4d: x = %b", $time, x);
#10 x = 1; $display("%4d: x = %b", $time, x);
#10 x = 1; $display("%4d: x = %b", $time, x);
#10 x = 0; $display("%4d: x = %b", $time, x);
#10 x = 0; $display("%4d: x = %b", $time, x);
#10 $finish;
end
endmodule

```

206

仿真输出

下面将展示 Moore FSM 的功能仿真输出结果。在仿真时间为 55 ~ 75 的时间段，Moore 信号 z 变为 1，意味着在测试向量中有两个“101”序列。

```

Chronologic VCS simulator copyright 1991-2009
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12;

```

```

0:          z = 0
10: x = 1
20: x = 1
30: x = 1
40: x = 0
50: x = 1
55:          z = 1
60: x = 0
65:          z = 0
70: x = 1
75:          z = 1
80: x = 1
85:          z = 0
90: x = 0
100: x=0
$finish called from file "tester.v", line 32.
$finish at simulation time      110

```

例 5-10 下面给出例 5-2 中检测重叠序列“101”的 Mealy 序列识别器的 Verilog 行为建模，其中代码由下面两部分组成：

第 1 部分代码：包含 NSG 和 OG 的行为描述。然而，值得注意的是，在一些大型设计中，组合的代码可能会存在同步问题，特别是使用了带有硬件源约束的逻辑器件时。这部分代码描述了 Mealy 有限状态图。有限状态图仅包含三个状态，分别被标识为 A、B 和 C。

第 2 部分代码：带有异步复位功能触发器的行为描述。reset 有效时，FSM 被初始化为初始状态 A。

HDL 建模

```
//A Mealy sequence recognizer that detects the overlapping
//sequence "101"
//Using binary encoded state labels
```

```
module mealy_seq
```

```
(
```

```
    input clock, reset, x,
```

```
    output reg z
```

```
);
```

```
parameter    A = 2'b00,
```

```
             B = 2'b01,
```

```
             C = 2'b10;
```

```
reg [1:0] current_state, next_state;
```

```
//Section 1: A combined next state generator (NSG) and
output generator (OG)
```

```
//unknown states are ignored
```

```
always@(*)
```

```
begin
```

```
    casex(current_state)
```

```
    A:  if (x == 1) begin
```

```
        next_state = B;
```

```
        z = 0;
```

```
    end
```

```
    else begin
```

```
        next_state = A;
```

```
        z = 0;
```

```
    end
```

```
    B:  if (x == 1) begin
```

```
        next_state = B;
```

```
        z = 0;
```

```
    end
```

```
    else begin
```

```
        next_state = C;
```

```
        z = 0;
```

```
    end
```

```
    C:  if (x == 1) begin
```

```
        next_state = B;
```

```
        z = 1;
```

```
    end
```

```
    else begin
```

```
        next_state = A;
```

```
        z = 0;
```

```
    end
```

```
    default: begin
```

```
        next_state = 2'bxx;
```

```
        z = 1'bx;
```

```
    end
```

```
endcase
```

```
end
```

207

208

```
//Section 2: flip-flops
always@(posedge clock, posedge reset)
begin
    if (reset == 1)
        current_state <= A;
    else
        current_state <= next_state;
end
endmodule
```

仿真测试台

测试平台和例 5-9 中所给的基本一样，除了“mealy_seq.v”文件已被实例化替代。

仿真输出

接下来给出 Mealy FSM 的功能仿真输出。注意在这种情况下，Mealy 输出 z 取决于当前状态和输入 x 。如果 x 发生变化， z 也可能变化。另一方面，在 Moore 机器中，输入 x 会在下个时钟周期影响 Moore 输出 z 。而这里，仿真时间在 50 ~ 70 之间，目标序列“101”的最后一位进入的同时，Mealy- z 输出 1。

```
Chronologic VCS simulator copyright 1991-2009
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12;
```

```
0:          z = 0
10: x = 1
20: x = 1
30: x = 1
40: x = 0
50:          x = 1
50:          z = 1
55: z = 0
60: x = 0
70: x = 1
70:          z = 1
75:          z = 0
80: x = 1
90: x = 0
100: x = 0

$finish called from file "testmealy.v", line 32.
$finish at simulation time                110
```

209

综合和仿真

通过使用 Altera Quartus II、ModelSim 和仿真工具，例 5-10 中 Mealy 序列识别器的 Verilog 建模进行了综合和仿真。图 5-41 为综合后的电路，图 5-42 为它的仿真波形。如时序图中所示，每次 x 的输入表示为一个“101”序列时，信号 z 就会变为 1。

有 2-1 选择器是可知的。

- 5.5 参考练习 5.2，这次直接通过一个 4 位的 4-1 选择器和一个 4 位的并行加载寄存器设计一个 4 位的寄存器。
- 5.6 设计一个带有异步复位的 8 位的多功能寄存器，能够表现出并行加载、循环右移或循环左移。
- 5.7 构建一个真值表验证图 5-16 中电路工作正常，表中包含当前状态位 q_1 、 q_0 ，输入 x ，下一状态位 d_1 和 d_0 ，输出 z 。复位之后，触发器初始化为 $q_1 = 0$ ， $q_0 = 0$ 。表中输入这些值作为它们 FSM 的当前状态。然后，设置 $x = 1$ ，使用公式 (5-2) 中给出的表达式确定 d_1 、 d_0 和 z 的值。把这些值填入表中。假设时钟信号发生 0-1 变化，改变 q_1 、 q_0 的值为 d_1 、 d_0 的值。在表中输入 q_1 、 q_0 的新值，在 x 的下个值为 1、0、1、0、1、0 时不断重复这个过程。从 z 的值中，确定电路是否工作正常。
- 5.8 设计一个 Moore 序列识别器检测非重叠的序列“101”。使用二进制编码状态标识画出电路示意图，要与图 5-16 中所示的相似。
- 5.9 设计一个 Mealy 序列识别器检测非重叠的序列“101”。使用二进制编码状态标识画出电路示意图，要与图 5-16 中所示的相似。
- 5.10 设计一个 Moore 序列识别器检测重叠序列“1001”。使用二进制编码状态标识。
- 5.11 设计一个 Mealy 序列识别器检测重叠序列“1001”。使用二进制编码状态标识。
- 5.12 参考图 5-13 中的有限状态图。使用二进制编码状态标识 11、01、10 和 00 分别标识状态 A ~ D。画出电路图，确定复位后，FSM 开始于状态 11 (A)。与图 5-15 和图 5-17 比较它们组合电路的大小。
- 5.13 设计一个 Mealy 序列识别器检测重叠序列“1001”。使用独热码状态标识并使用 Espresso 最小化组合的真值表。
- 5.14 使用 D 触发器设计一个形式上的 JK 触发器（也可以参考第 4 章）。
- 5.15 仿真带有如下要求的 Verilog 建模的电路：
- 设计例 5-1 中的 FSM，但使用公式 (5-2) 中给出的表达式。
 - 直接通过描述有限状态图的方式设计例 5-1 中的 FSM。
- 5.16 参考图 5-22 中的有限状态图。使用二进制编码标识设计相关的 FSM，与图 5-15 中的电路比较电路大小。
- 5.17 设计一个带有低电平有效的异步复位的位串行模 11 计数器（也可以参考图 5-28）。
- 5.18 设计一个带有低电平有效的异步复位的位并行模 11 计数器（也可以参考图 5-31）。
- 5.19 仿真带有如下要求的 Verilog 电路建模：
- 为图 5-28 所给的 1 位二进制计数器片进行建模，然后用它设计练习 5.17 中的计数器。
 - 使用行为建模的加法器、选择器和并行加载寄存器，用它们去设计练习 5.18 中的计数器。
 - 例 5-18 中计数器的完整行为建模。
- 5.20 设计一个带有低电平有效的异步复位的模 4 上/下计数器（非计数器片）。
- 5.21 设计一个同时带有同步和异步复位功能的模 4 上/下计数器（非计数器片）。
- 5.22 设计一个同时带有同步和异步复位信号的 3 位格雷码计数器。
- 5.23 反相不归零 (NRZI) 是一种用于与通用串行总线 (USB) 设备通信的数据编码方案。NRZI 发生器的输出信号 (z) 在输入位 (x) 是 0 时发生转变，在输入位是 1 时保持原来的值 (0 或 1) 稳定不变。也就是说，从右到左，当 NRZI 发生器的输入是 000000 时，它的输出从右到左将会转换，为 101010。然而，输入中连续的 1 对应的输出将保持它之前的值。例如，输入 X 从右到左为：1111000111000011，NRZI 发生器的输出 z 从右到左为：0000010111101

0 1 1。类似地, $X = 0xCF0C$ 时, $Z: 0xEFAE$ 。设计这种 NRZI 发生器。

- 5.24 假设一个有限状态图有 5 种状态。使用汉明编码方案产生 5 个汉明码去标识这些状态。每一对标识码应当有 3 个或者更多的汉明间距。
- 5.25 创建并仿真例 5-7 中容错 FSM 的 Verilog 建模。特别地, 复制表 5-12 的条目到一个 Excel 表中。然后在每一行中使用 Excel 的“连接”功能将位 $q_4 \sim q_0$ 和 x 连接为一个 6 位的二进制数。对 $d_4 \sim d_0$ 和 z 做同样的处理。(你也可以使用 Excel 的 bin2hex 功能将这个连接的 6 位数转换为十六进制的 6 位数)。然后对这个表进行排序, 使得这个表中的行按照当前状态位和 Excel 表的 x 位有序上升排列。然后创建两列, 使用连接功能将两个 6 位的条目转换为正确的 Verilog 语句声明, 用在 case 语句中。例如, 表示 q 和 x 的 000000 写为 “6'h000000:”, 表示 d 和 z 的 000000 写为 “{d, z} = 6'b000000:”, 其中 d 在 Verilog 中被声明为一个 5 位的下一状态变量。从 Excel 中复制这两列到一个 Verilog 文本编辑器中, 用组合建模为 NSG 和 OG 模块的 FSM 进行建模。为了简化, 建模触发器时使用带有普通复位但有独立置位的信号。在这个测试台中, 使用置位信号产生一个 1 位的错误; 也就是, 改变 $q = 0$ 为 1。没有其他错误时 FSM 应当可以继续工作。

212

```
...
always@(*) //NSG and OG
begin
case({q, x}) //q is declared as 5-bit current state variable
6'h000000: {d, z} = 6'h000000; //1st row of the table
...
default: {d, z} = 6'hx;
endcase
end
```

- 5.26 设计一个容错的模 4 向上计数器(非计数器片)。
- 5.27 参考图 5-15 中所给电路。在不改变它的组合电路的情况下, 使用汉明错误检测机制, 在电路中添加模块, 让它工作时表现为一个一位容错 FSM。
- 5.28 一个串行加法器输入 2 位的 x 和 y , 每个时钟周期输出它们的和位 s 。它内部保持外部输出位的存在或不存在。假设初始的进位为零, 完成以下内容:
- 画一个串行加法器的 Mealy 有限状态图。
 - 设计 Mealy 串行加法器的 FSM。
 - 画一个串行加法器的 Moore 有限状态图。
 - 设计 Moore 串行加法器的 FSM。
 - 设计一个容错的 Mealy 串行加法器。
- 5.29 参考图 5-34 中的电路, 假设 $\Delta_{CC} = 0.3ns$, $\tau_{sc} = 0.2ns$, $\tau_{st} = 0.05ns$, $\tau_{cq} = 0.05ns$ 。假设 $\tau = 0.6ns$, 画一个时序图并讨论电路工作中是否会因为时钟相位差出现问题。
- 5.30 参考图 5-36 中的电路, 假设 $\Delta_{CC} = 0.3ns$, $\tau_{sc} = 0.2ns$, $\tau_{st} = 0.05ns$, $\tau_{cq} = 0.05ns$ 。假设 $\tau = 0.6ns$, 画一个时序图并讨论电路工作中是否会因为时钟相位差出现问题。
- 5.31 参考图 5-36 中的电路, 假设 $\Delta_{CC} = 0.3ns$, $\tau_{sc} = 0.2ns$, $\tau_{st} = 0.05ns$, $\tau_{cq} = 0.05ns$ 。假设 $\tau = 0.7ns$, 画一个时序图并讨论电路工作中是否会因为时钟相位差出现问题。
- 5.32 参考一个硬件先入先出缓冲区, 其输入输出端口连接到了并行计算的时序电路 A 和 B 上 [4-5]。电路 A 产生 IN, 作为写一个值的下一个缓冲区, 电路 B 产生 OUT, 作为读一个值的下一个缓

缓冲区。两个计数器，X 和 Y，每一个被其中的一个电路控制，分别用来产生 IN 和 OUT 值。缓冲区工作在一个循环的方式中。这两个电路，还有它们分别的计数器，工作在两个不同的时钟下。电路 A 和 B 也需要输入 IN 和 OUT 值，去确定缓冲区是满还是空。完成以下内容：

213

- a. 画出电路 A 和 B、计数器、带标识的缓冲区的框图，还有信号的名字。IN 是电路 B 的异步输入，OUT 是电路 A 的异步输入。为每一位使用两个同步触发器。
- b. 假设缓冲区大小是 8，X 和 Y 被设计为了模 8 计数器。假设 IN 是 3，列出电路 B 的输入 IN 经过同步触发器的可能的值。讨论电路 B 的输入 IN 不是 3 时，缓冲区的空标志将如何产生。
- c. 重复 (a) 部分，但这次假设 X 和 Y 被设计为了格雷码计数器。

计算机安全

- 5.33 计算机安全（硬件木马）：查看练习 11.12，了解单输入触发恶意电路计算（也可以看 11.2 节）。
- 5.34 计算机安全（硬件木马）：查看练习 11.13，了解计时器恶意攻击电路计算（也可以看 11.2 节）。
- 5.35 计算机安全（加密）：查看练习 11.14，设计一个硬件加密电路（也可以看 11.5 节）。
- 5.36 计算机安全（计算机安全威胁）：查看练习 11.15，了解计算机安全威胁（也可以看 11.1.3 节）。
- 5.37 计算机安全（硬件发展的威胁）：查看练习 11.16，了解同态加密硬件发展安全策略机制（也可以看 11.1.3 节和 11.2 节）。

214

时序电路：大型设计

6.1 简介

如图 6-1 所示，一个大的时序电路由数据通路和控制单元设计而成。数据通路同时包括时序和组合的电路模块，如寄存器、计数器、选择器（MUX）、译码器、算术逻辑单元（ALU）和其他等，它们是标准的或特定的问题。这些模块集体地实现一系列简单的操作，比如把两个寄存器的内容相加并把结果存在第三个寄存器中。控制单元负责为提供操作的数据通路提供必要的控制信号。一个数据通路在一个时钟周期中表现出一种或多种操作。

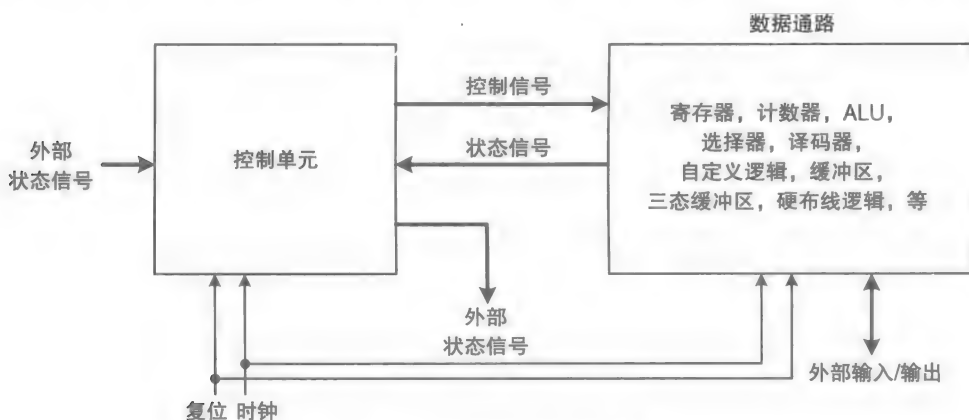


图 6-1 大型时序电路的框图

每个数据通路操作需要一个或多个数据输入，产生一个或多个输出，然后其中每一个又被存储在了寄存器或内存中。一个操作可能也会使用一个或多个组合电路单元来计算得到一个输出。在数据通路中，两个或多个操作可能共享部分或全部的组合电路模块。例如，从不同源中读取数据计算和的两次操作可以通过共享同一个加法器来减少硬件。

一个数据通路也可能条件性地表现出一个或多个操作，取决于信号的值是来自内部的数据通路或由数据通路外部的另外一个模块产生的。一个被称为算术溢出标志的特定寄存器位值和一个特定的计数值是数据通路内部信号的实例，这代表了一种情况。外部事件触发信号，如启动控制单元的一个信号，按键产生的一个信号，表示一个内存数据有效可读的一个信号，都是表示数据通路外部信号的条件实例。

有几种可选的结构来设计数据通路和控制单元。许多因素可以影响数据通路的结构，包括操作时钟的频率和应当产生结果的频率。一个较高的时钟频率表示了数据通路要有更短的最大传播延时，这个决定了结果产生的有多快。然而，一个电路的时钟频率和电路消耗功率的晶体管的数量之间有一定的关系，这个决定了电路可能释放出多少热量。如第 1 章中提到的，一个时序电路在使用风扇冷却系统时仍在允许的温度范围内时可以操作多快有一个限制。在实践中，使用当前的芯片技术，集成电路平均可以释放的热量限制了它的时钟频率可

以有多高，它可以包含多少个晶体管。

在这一章中，我们使用不同的数据通路和控制单元结构，解释它们的组织形式，评估它们的性能参数。我们还使用复杂时序电路的模块来表示功率和能量，讨论这些模块使用时如何减少电能消耗，还有如何估计复杂时序电路的能源效率。

寄存器传输符号

寄存器传输符号（RTN）用于正式描述数据通路的操作路径。每一个操作产生一个结果，其必须存储在存储模块中，比如寄存器或内存。RTN 的语法是随意的。例如， $R3 \leftarrow R1 + R2$ 是一个 RTN，它包括了三个寄存器， $R1$ 、 $R2$ 和 $R3$ 。左箭头（ \leftarrow ）表示在下一个时钟周期中两个寄存器值的和将被存储在寄存器 $R3$ 中。表 6-1 表示了书中用到的一些 RTN 语法示例。部分语法取自于 Verilog 硬件描述语言中（HDL）。

表 6-1 使用随意语法和 Verilog 硬件描述语言语法的 RTN 语法举例

| RTN 举例 | 含 义 |
|---|---|
| $R \leftarrow value \quad \rightarrow$ | 描述一个寄存器加载，寄存器 R 装载这个值 |
| $CNTR \leftarrow CNTR + 1$ 或 $CNTR \leq CNTR + 1; \text{ (Verilog)}$ | 描述一个名为 $CNTR$ 的计数器增值 |
| $R \leftarrow 0 \quad //R[7..1]$ 或 $R \leq R \gg 1; \text{ (Verilog)}$ 或 $R \leq \{0, R[7:1]\}; \text{ (Verilog)}$ | 描述一个带有 0 填充的寄存器右移。符号 “//” 这里用来表示连接 |
| $R \leftarrow R[7] \quad //R[7..1]$ 或 $R \leq R \ggg 1; \text{ (Verilog)}$ 或 $R \leq \{R[7], R[7:1]\}; \text{ (Verilog)}$ | 描述一个算术右移 |
| $R3 \leftarrow R1 + R2$ 或 $R3 \leq R1 + R2; \text{ (Verilog)}$ | 描述将两个寄存器 $R1$ 和 $R2$ 的值相加，并将和保存在 $R3$ 中 |
| $M[X] \leftarrow R;$ | 描述一个内存传输。寄存器 R 的值存储在内存位置 X 处 |

6.2 数据通路设计

一个数据通路的结构可以被分类为单周期、多周期或流水线。单周期的数据通路需要更多的硬件，但只需简单的控制单元。多周期的数据通路需要更少的硬件，但使用多个时钟周期按步产生结果。流水线数据通路也需要更多的硬件，但同时可以操作多个输入。流水线只有在多个输入的情况下处理起来才会有效率。

例 6-1 表现一个或多个 $R \leftarrow A + B + C \pm D$ 类型的 RTN 操作的单周期、多周期和流水线数据通路的设计和表现在这里表示了出来。 A 到 D 代表了同时从一些寄存器或内存中读取到的值。然而，流水线数据通路将会表示为 $R_i \leftarrow A_i + B_i + C_i \pm D_i$ ，其中 $i = 0, 1, 2, 3$ 等。

RTN $R \leftarrow A + B + C + D$ 和 $R \leftarrow A + B + C - D$ 不代表标准 CPU 指令所表现的操作, 我们将会在第 8 章中得知。这些 RTN 被认为是混合操作, 需要在计算上使用三个或更多的数据。一个典型的算术指令操作使用两个数据值。这里, RTN 用来解释和比较单周期、多周期和流水线数据通路设计。然而, 一些 CPU (比如 [1]) 有一些使用三个数据值表现出混合操作的指令, 比如乘-加 ($R \leftarrow A + B * C$), 这是一个计算中常用的操作, 包括矩阵。如果 A 、 B 和 C 是浮点 (FP) 数, 混合操作会有在内存中产生结果时只有一个舍入误差的优点 (见 3.8.3 节)。要是分开在两条指令中完成, $B * C$ 的结果如果存储在内存中, 将会产生一次舍入误差, 然后用 A 加上这个内存中值产生的结果, 如果也放在内存中, 将会产生另外一个舍入误差。

其他的例子包括自定义指令执行混合操作提出了可配置 CPU 的设计 [2]。在这种情况下, 在一个程序循环中表现为一系列指令序列的独立操作可以合为一个带有混合操作的单一的自定义的指令。这个新的指令代替了循环中的指令序列, 这样通过减少必须从内存中读取的指令的数量来提高性能。

第 1 章中讨论过的 SIMD 架构是另一种操作多个数据值的指令。然而在这种情况下, 每一条 SIMD 指令只指定一种操作, 这种操作会同时发生而且没有依赖多个数据值的数据。这三个数据通路的每一种控制单元的设计将在 6.4 节中进行讨论。

6.2.1 单周期

图 6-2 表示了计算值 $A + B + C + D$ 或 $A + B + C - D$ 而且把结果在一个时钟周期内存储在寄存器 R 的单周期的数据通路。这个数据通路包括两个加法 (+) 模块和一个加/减 (+/-) 模块。信号 $mode$ 控制加/减模块的功能。如果 $mode = 0$, 数据通路表现为 $R \leftarrow A + B + C + D$; 否则, 它将表现为 $R \leftarrow A + B + C - D$ 。

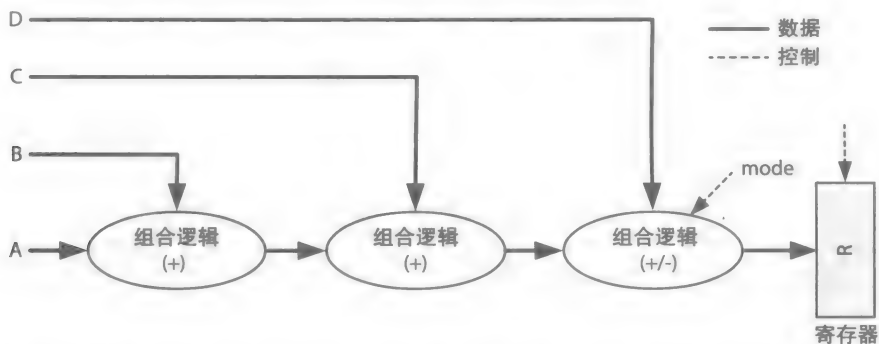


图 6-2 在一个时钟周期内计算 $A + B + C + D$ 或 $A + B + C - D$ 的单周期的两个功能的数据通路

公式 (6-1) 用来估计运行这个数据通路所需的最小时钟周期。这个周期与最长的信号路径的传播延时成正比, 信号路径从进入第一个加法器开始计算, 到进入这个寄存器结束。

$$\tau \geq 2\Delta_{\text{ADD}} + \Delta_{\text{ADD/SUB}} + \tau_{\text{st}} + \tau_{\text{cq}} + \tau_{\text{cs}} \quad (6-1)$$

通常, 如果一个单周期的数据通路实现几个单一或复杂的操作, 它的最小时钟周期将正比于完成最复杂的操作所需的时间。因此, 简单和复杂的操作将都会需要同样的时间来完成。这将会增加总的时间来完成同时需要简单和复杂操作的任务。

6.2.2 多周期

一个多周期的数据通路需要把一种计算进行分解并分步完成，每一步需要一个简单的数据通路操作。图 6-3 描述了一个带有简单加法器 / 减法器 和两个选择器 (MUX) 模块的多周期数据通路。这个数据通路能表现出 5 种可能的简单操作，比如 $R \leftarrow A$, $R \leftarrow R + B$, $R \leftarrow R + C$, $R \leftarrow R + D$ 或 $R \leftarrow R - D$ 。下面的算法使用了 4 个周期实现了 $R \leftarrow A + B + C \pm D$ 。

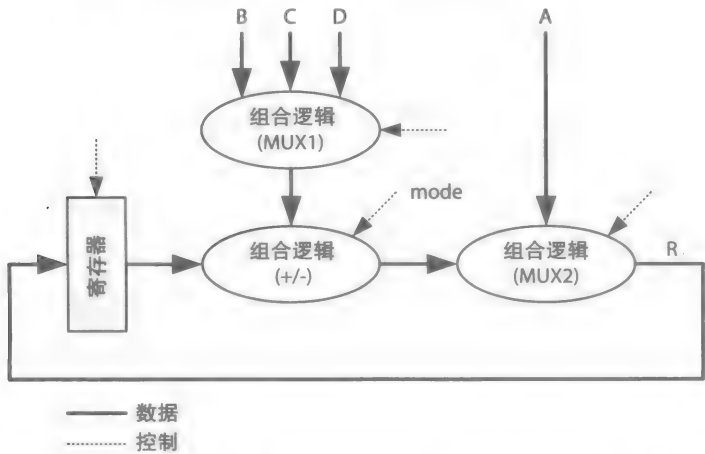


图 6-3 需要 4 个时钟周期来计算 $A + B + C + D$ 或 $A + B + C - D$ 的多周期数据通路

实现 $R \leftarrow A + B + C \pm D$ 的一个多周期算法：

- 周期 1: $R \leftarrow A$
- 周期 2: $R \leftarrow R + B$
- 周期 3: $R \leftarrow R + C$
- 周期 4: 如果 $mode == 0$, $R \leftarrow R + D$; 否则, $R \leftarrow R - D$

多周期数据通路的时钟周期也和最长信号路径的时延成正比。在这种情况下，最长的路径从穿过加法器 / 减法器模块的 MUX1 的输入开始，到寄存器的输入结束。公式 (6-2) 估算了这个数据通路的最小时钟周期。注意 MUX 的传播时延比一个加法器的小。因此，估算的这个多周期数据通路的最小时钟周期比单周期数据通路的小。然而，这个多周期算法需要 4 个时钟周期来完成这个任务，而单周期数据通路只需要一个时钟周期。

$$\tau \geq \Delta_{MUX1} + \Delta_{ADD/SUB} + \Delta_{MUX2} + \tau_{ff} + \tau_{cq} + \tau_{cs}$$

(6-2)

一个多周期数据通路有一个减少了所需硬件总量的优点。在这个图中，一个简单的加法器 / 减法器要被使用多次来产生最终的结果。通常，如果简单和复杂的计算都要实现，一个多周期的数据通路也会有一个优势。在这种情况下，完成一个简单的计算会需要较少的时钟周期数，完成一个复杂的计算会需要较多的时钟周期数。因此，它将会减少总的所需时钟周期数来完成一个任务。另外，与一个单周期的数据通路形成对比，一个多周期的数据通路将需要一个更高频率（短周期）的时钟。从另一方面，一个单周期的数据通路将需要一个更慢（长周期）的时钟，但仅需要一个时钟周期来完成每一个简单或复杂的计算。

6.2.3 流水线

一个流水线数据通路，或**流水线**，是处理数据流的理想结构。例如，考虑使 N 对 FP 数

字相加，每次一对，来产生 N 个和，或者考虑执行 N 个汇编指令。当以流水线方式处理，一次计算被分为一系列独立的操作，其中每个在被称为**流水线阶段**的单独的子数据通路中进行，更像是多周期数据通路中的那样。

这个阶段不共享任何模块，并由并行加载寄存器形成一个流水线，就像第1章中讨论过的汽车制造生产线。所有的阶段并发操作来处理数据流。比如，考虑计算 N 个值的问题，从 $i = 0$ 到 $N - 1$ 中的每一个 $A_i + B_i + C_i \pm D_i$ 。每一个 A_i 到 D_i 代表了4个数据项，比如4个不同数组中的第 i 个元素。分离每一种计算 $A_i + B_i + C_i \pm D_i$ 的方式的组相关操作如下：

图 6-4 表示了一个标有 1 ~ 3 三个阶段的流水线数据通路的结构。三组寄存器用来分离每一个阶段产生的结果。在这个图中，这个流水线从一个外部模块接收到一系列的4个值 A_i 、 B_i 、 C_i 和 D_i ，也向一个外部模块发送这个结果 R_i 。在每一个时钟周期中，阶段1从一个外部数据源获取它的输入，阶段2从阶段1中获取它的输入，阶段3从阶段2中获取它的输入。因此，在每一个时钟周期中，所有的三个阶段并行操作。

$$\begin{aligned} X_i &\leftarrow A_i + B_i && // \text{流水线阶段 1 的操作} \\ Y_i &\leftarrow X_i + C_i && // \text{流水线阶段 2 的操作} \\ R_i &\leftarrow Y_i \pm D_i && // \text{流水线阶段 3 的操作} \end{aligned}$$

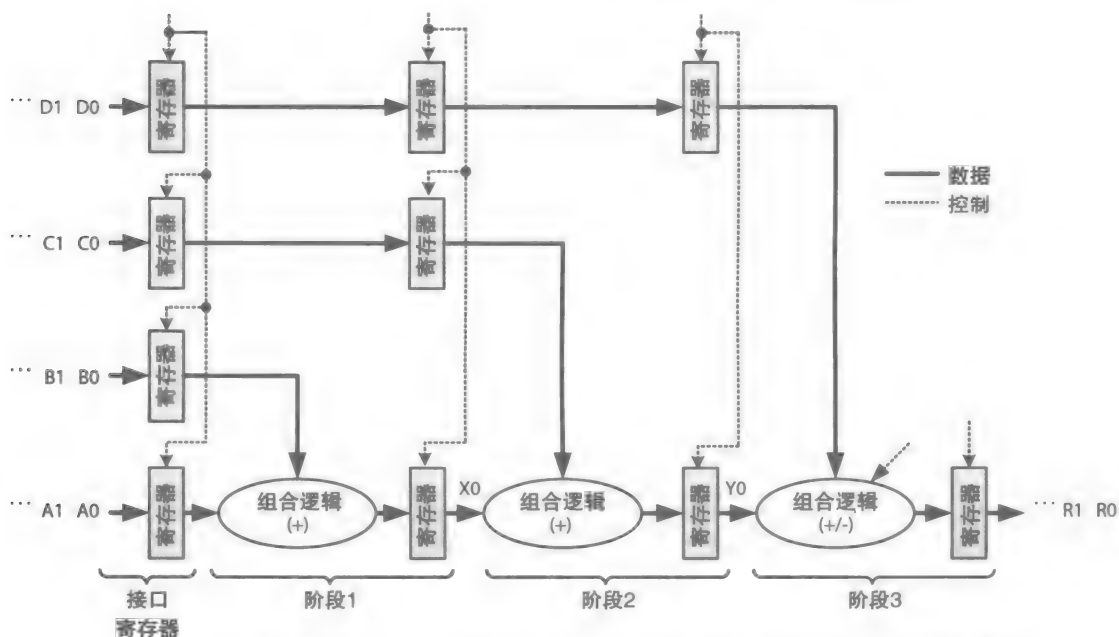


图 6-4 计算一系列 $i = 0, 1, 2$ 等的 $A_i + B_i + C_i \pm D_i$ 值的双功能的流水线数据通路

图 6-5 给出了表示流水线的两种不同的**流水线图表**类型。注意图表不包括图 6-4 中所示由接口寄存器造成的一个周期的延时。图 6-5a 中的流水线图表有一个水平方向的组织，在 x 轴上表示出了时钟周期。另一方面，图 6-5b 中的图表有一个垂直方向的组织，在负 y 轴上表示了时钟周期。

如图 6-5 所示，在周期 3 中，当阶段 3 正在产生 R_0 ，阶段 2 正在产生 R_1 所需要的中间结果 Y_1 ，阶段 1 正在产生 R_2 所需要的中间结果 X_2 。因此，流水线同时表现出了三种简单的操作，如此在多个数据值上并行操作。这有助于快速完成任务。

| | | | | | |
|-----|----------------------------|----------------------------|------------------------------|------------------------------|-----|
| 阶段3 | | | $R_0 \leftarrow Y_0 \pm D_0$ | $R_1 \leftarrow Y_1 \pm D_1$ | ... |
| 阶段2 | | $Y_0 \leftarrow X_0 + C_0$ | $Y_1 \leftarrow X_1 + C_1$ | $Y_2 \leftarrow X_2 + C_2$ | ... |
| 阶段1 | $X_0 \leftarrow A_0 + B_0$ | $X_1 \leftarrow A_1 + B_1$ | $X_2 \leftarrow A_2 + B_2$ | $X_3 \leftarrow A_3 + B_3$ | ... |
| 周期 | 1 | 2 | 3 | 4 | ... |

a) 水平方向组织

| | | | |
|-----|----------------------------|----------------------------|------------------------------|
| 周期 | 阶段1 | 阶段2 | 阶段3 |
| 1 | $X_0 \leftarrow A_0 + B_0$ | | |
| 2 | $X_1 \leftarrow A_1 + B_1$ | $Y_0 \leftarrow X_0 + C_0$ | |
| 3 | $X_2 \leftarrow A_2 + B_2$ | $Y_1 \leftarrow X_1 + C_1$ | $R_0 \leftarrow Y_0 \pm D_0$ |
| 4 | $X_3 \leftarrow A_3 + B_3$ | $Y_2 \leftarrow X_2 + C_2$ | $R_1 \leftarrow Y_1 \pm D_1$ |
| 5 | $X_4 \leftarrow A_4 + B_4$ | $Y_3 \leftarrow X_3 + C_3$ | $R_2 \leftarrow Y_2 \pm D_2$ |
| ... | ... | ... | ... |

b) 垂直方向组织

图 6-5 两个可选的流水线图表：a) 从左到右；b) 从上到下；表示了结果 R_0 、 R_1 ，等等

流水线使用更多的硬件，类似于一个单周期数据通路，但工作在一个高频率的时钟下，这点类似于多周期数据通路。而且，它能比其他两个数据通路更快地处理一系列的数据。一个流水线数据通路的时钟周期正比于它的最长阶段的传播时延。在图 6-4 中，阶段 3 有最长的传播时延；它使用了一个加法器 / 减法器模块，而其他两个阶段使用了一个加法器模块。公式 (6-3) 估算了流水线的时钟周期。

221

$$\tau \geq \Delta_{\text{ADD/SUB}} + \tau_{st} + \tau_{cq} + \tau_{cs} \tag{6-3}$$

图 6-4 中的流水线数据通路被称为**线性流水线**，它的三个阶段中的每一个只用了一次来计算最终的结果（比如 R_0 ）。从另一方面，**非线性流水线**使用它的一个或多个阶段多次来计算出最终的结果。非线性流水线的设计与其他的東西有关。

一个时钟周期中，线性流水线的每一个阶段都输入来自其前面紧相连阶段的一个或多个数据项。通常情况，第一个阶段的数据要么从外部模块读入，要么来自内部存储器。最终的结果要么存储在一个阶段中的存储模块（寄存器或内存）中，要么发送到一个外部模块上。这些将在第 8 章中进一步讨论。

流水线性能

图 6-5 中的流水线图表 a 或 b 的学习揭示了作为第一个结果的 R_0 需要三个时钟周期来计算（不包括第一个所需的时钟周期来加载接口寄存器），而结果 R_1 、 R_2 等，每一个只需要一个时钟周期来计算出。这样减少了总的时间来计算 N 个最终的结果。通常情况下，一个 k 阶段（线性）的流水线需要 k 个时钟周期来产生它的第一个输出。公式 (6-4) 估算了使用一个 k 阶段线性流水线来处理个数为 N 的数据流所需的总时间，其用时钟周期 τ_{pipeline} 表示。

$$T_{\text{pipeline}} = k * \tau_{\text{pipeline}} + (N - 1) \tau_{\text{pipeline}} \tag{6-4}$$

例如，当 $N = 3$ ， $k = 3$ 时，这种流水线将共需要 5 个 τ_{pipeline} 来产生三个输出，比如图 6-5b 所示的三个输出 R_0 、 R_1 和 R_3 。假设一个单周期数据通路的时钟周期 $\tau_{\text{single-cycle}}$ 大约等于 $k * \tau_{\text{pipeline}}$ ，其中 τ_{pipeline} 是相关流水线的时钟周期，公式 (6-5) 估算了使用单周期数据通路处理个数为 N 的数据流所需的总的时间。

$$T_{\text{single-cycle}} = N * k * \tau_{\text{pipeline}} \tag{6-5}$$

然而, 注意在通常情况下, 即使一个单周期数据通路和相应的 k 阶段流水线都使用带有不同传播时延的不同的组合电路, $\tau_{\text{single-cycle}}$ 将会稍小于 $k * \tau_{\text{pipeline}}$ 。 $k * \tau_{\text{pipeline}}$ 的值包括 k 个寄存器的建立时间和 k 个时钟到 q 的时延的和, 而 $\tau_{\text{single-cycle}}$ 将只包含一个寄存器的建立时间和一个时钟到 q 的时延的和。 $k * \tau_{\text{pipeline}}$ 也是一个最大值估计, 因为 τ_{pipeline} 与这个流水线的最长阶段的传播时延成正比。在图 6-4 中, τ_{pipeline} 是基于阶段 3 的传播时延计算出来的, 阶段 3 使用了一个加法器/减法器模块, 而其他两个阶段每一个使用了一个加法器。然而, 为了和相应的单周期数据通路相比时简化流水线的性能分析, $\tau_{\text{single-cycle}}$ 和 $k * \tau_{\text{pipeline}}$ 的近似值之间的不同被忽略了。

222

通常, 加速比是一种性能参数, 和一个较慢的系统进行对比完成同样的任务时测试较快的系统的性能。它被定义为一个较慢的系统与一个较快系统完成同一个任务所需时间的比例。它表示了和一个等效的较慢系统对比时, 一个较快的系统有多快。例如, 公式 (6-6) 定义了当处理个数为 N 的数据流, $\tau = \tau_{\text{pipeline}}$ 时的较快的流水线数据通路和对应的较慢的单周期数据通路对比时的加速比。 $N = 3, k = 3$ 时, 流水线比单周期数据通路大约快 1.8 ($3 * 3 * \tau_{\text{pipeline}} / 5 \tau_{\text{pipeline}} = 1.8$) 倍。 $N = 1000, k = 3$ 时, 加速比大约是 2.99。注意, 当 N 趋近于无穷大 (∞) 时, 加速比增加并趋近于 k (阶段的个数)。倘若处理一个较大的数据流, 流水线有越多的阶段, 加速比将会越大。当 N 变大时, 数据流的处理会更有效率, 因为和处理整个数据流所需的总时间相比, 填充管道所需的时间可以忽略不计。

$$\text{加速比} = \frac{T_{\text{single-cycle}}}{T_{\text{pipeline}}} = \frac{Nk\tau}{k\tau + (N-1)\tau} = \frac{Nk}{k + N - 1} \quad (6-6)$$

效率是用来测量完成一个任务时系统的资源分配有多合理的性能参数。如果一个流水线数据通路的所有阶段在任何时间都处于忙碌中, 我们就说它的效率是 100%。也就是说, 没有空闲的阶段。例如, 考虑图 6-5a 中的流水线图表。从第 3 个时钟周期开始, 当所有的阶段都在工作中处理剩下的计算, 这个流水线的效率达到 100%。然而, 我们需要整体的效率值, 而不是当所有的资源都在使用中时。从公式 (6-6) 中得知, 当计算量的个数 N 趋近于无穷大 (∞) 时, 流水线的加速比趋近于阶段的个数 K 。因此, 系统的整体效率可以定义为它的加速比与它可能的最大加速比的比率。公式 (6-7) 定义了 K 个阶段的流水线数据通路的效率。

$$\text{效率} = \frac{\text{加速比}}{k} \quad (6-7)$$

将公式 (6-6) 替换到公式 (6-7) 中,

$$\text{效率} = \frac{N}{k + N - 1}$$

当 N 趋近于无穷大 (∞) 时, 如所预期的, 图 6-4 中流水线的效率趋近于 1 或 100%。

吞吐量是测量一个系统处理速率的另一个性能参数。它代表了每秒钟执行的事件的数量 (N)。它被计算为执行的总的事件 (任务、计算量、操作、谷歌搜索等) 的数量与所需的总时间 (T) 之间的比率。公式 (6-8) 定义了带有 k 个阶段的线性流水线的吞吐量。对于 $N = 3, k = 3$, 吞吐量大约是 $0.6\tau^{-1}$ ($3/5\tau$)。对于 $N = 1000, k = 3$, 它大约是 $0.99\tau^{-1}$ ($1000/1002\tau$)。通常, 当 N 趋近于无穷 (∞) 时, 线性流水线的吞吐量趋近于 τ^{-1} (工作的时钟频率)。例如, 如果图 6-4 中的流水线的时钟频率 ($f = 1/\tau$) 是 1GHz (每秒十亿个周期), 它的峰值吞吐量 (τ^{-1}) 将会是十亿的计算量 (对每一种 $A + B + C \pm D$), 或者每秒三十亿的算术计算量 (对每一种 + 或 -), 不包括读取输入数据所需的延时, 例如, 从内存读取并将输出写回到内

223

存中。

$$\text{吞吐量} = \frac{N}{T_{\text{pipeline}}} = \frac{N}{k\tau + (N-1)\tau} \quad (6-8)$$

正如第 1 章中讨论过的，CPU 的数据通路是流水线的，因为它执行很多指令，包括操作浮点数的浮点型（FP）指令。对于一个实现浮点结构的数据通路，它必须表现出多种操作，比如第 3 章中讨论过的初始化、对齐小数点、整数算术、正规化和四舍五入。这些浮点数操作通常被分成几个流水线阶段来提高吞吐量。例如，考虑下面的 for 循环中一个浮点 ADD 指令（即 FADD），它将被执行 1000 次来把数组 A 中的 1000 个元素和数组 B 中的 1000 个元素相加，产生数组 C 中的 1000 个元素。在有流水线浮点单元（FPU）中，和单周期的 FPU 相比，这 1000 个加法指令执行时将会占用更少的时间。

```
float A[1000], B[1000], C[1000];
int i;
for(i = 0; i < 1000; i++)
    C[i] = A[i] + B[i];
```

FLOPS（每秒浮点运算速度）或较少使用的 MIPS（每秒执行的百万条指令）是处理器设计师通常报告的吞吐量测试单元的两个实例。然而，设计师通常报告的这些吞吐量单元经常是设计师假设的理想条件和可能代表峰值的性能值。另外，MIPS 可能基于执行一组随机的指令组合。通常，更现实的性能测试需要一些基准（现有的标准）程序的执行，比如测试计算机系统性能的称为标准性能评估公司 2006 年基准的计算密集型工作负载，或测试计算机图形系统性能的图形密集型工作负载的 SPECviewperf 基准 [3]。

6.3 控制单元设计技术

一个控制单元是一个有限状态机（FSM）。作为一个硬件控制单元，控制信号由一系列的组合电路产生。为了实现最大的速率，每一个控制信号可以是带有最大值 $3\Delta_{\text{NAND}}$ 或 $3\Delta_{\text{NOR}}$ 的与或（SOP 表达式）或或与（POS 表达式）电路的输出。

224

一个控制单元可以作为一个有限状态图（FSD）被建模或使用之前章节中讨论过的位并行方式来设计。然而，一旦一个硬件控制单元被创建了，如果它有设计错误，将不可修复，尤其是如果它实现一系列非常复杂的算术。一个高性能的流水线数据通路通常由一个硬件控制单元控制。

另一方面，被称为微程序控制的基于内存的控制单元，在 IC 的内存中保存控制信号的值。万一将来手工地将一些设计错误解决了，内存中的内容可以被更新。然而一个基于内存的控制单元可能较慢，取决于内存的大小。它使用远大于 $3\Delta_{\text{NAND}}$ 的时间来完成一次内存读/写操作，这将在第 7 章中进行讨论。

微程序控制的应用几年来已经减少，尤其因为与复杂指令集计算机（CISC）结构形成对比的精简指令集计算机（RISC）的优势。与 CISC 相比，对于 RISC，CPU 拥有更简单、更少的指令。因此，很容易设计出一个硬件控制单元来控制一个 RISC 数据通路。RISC 和 CISC 将会在第 8 章中进行讨论。

微程序控制应用的减少也因为现在有现代 HDL 综合工具的可用性。这个工具已经简化了硬件控制单元的设计和验证。然而，当设计一个含有较大数量状态的控制单元时，或者当有必要将一个传统的 CISC 指令翻译成一系列简单的操作并通过一个更有效的 RISC 数据通路来实现时，微程序控制将仍然被用到 [4]。

6.3.1 硬件控制单元：FSD

图 6-6 表示了图 6-3 中给出的多周期数据通路，并带有它的控制单元的 FSD 模块。这个 FSD 有 4 个状态，并用 RTN 定义了数据通路的操作。这个控制单元产生数据通路的控制信号，当外部输入信号 $mode = 0$ 时在 4 个时钟周期内来计算 $A + B + C + D$ 的值，或当外部输入信号 $mode = 1$ 时在 4 个时钟周期内来计算 $A + B + C - D$ 的值。这个结果将被存储在数据通路的寄存器中。当插入了外部输入信号 $start$ 后，会触发计算的启动。相反，图 6-7 描述了带有实际控制信号的 FSD。注意，数据通路的操作由 RTN 指定比由实际的控制信号来指定更容易鉴别一个 FSD。

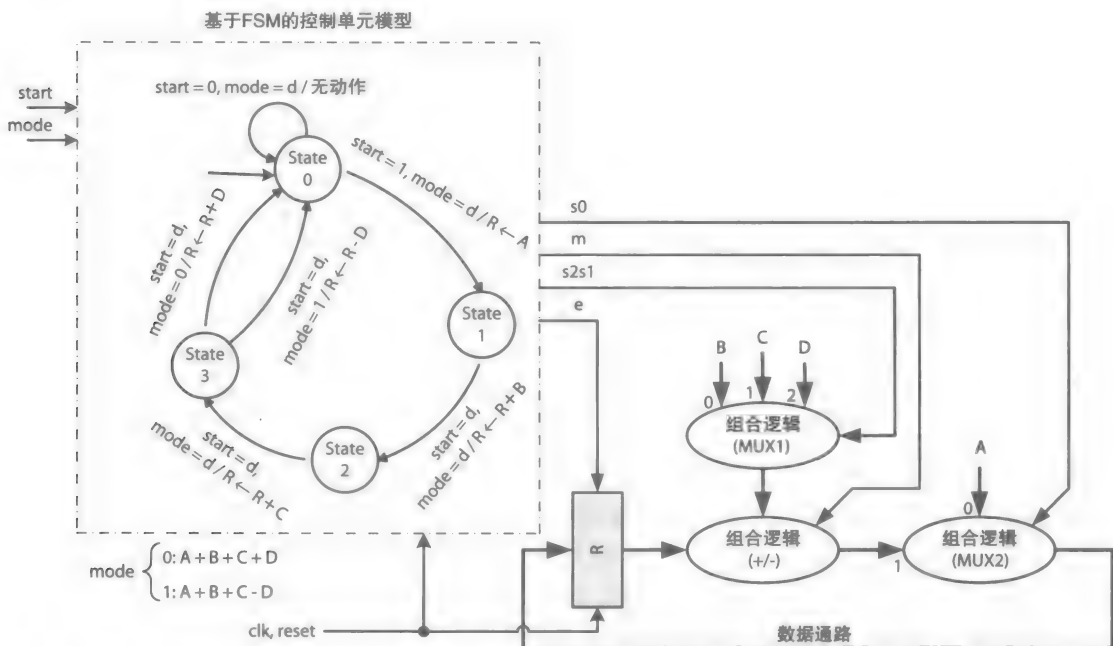


图 6-6 解释图 6-3 中由 RTN 表示的数据通路的基于 FSM 的控制单元，其中“d”表示不关心

图 6-8 描述了带两个触发器：下个状态发生器 (NSG) 和输出发生器 (OG) 的相应的 FSM 的详细框图。OG 负责产生数据通路的控制信号。在每一个时钟周期中，只有与指定的数据通路操作相关的控制信号才会被插入。例如，最初时，当 $start = 0$ 时，寄存器 R 被关闭，数据通路被认为“无动作”。当 $start$ 变为 1，控制单元插入 e (即 $e = 1$)，使能了寄存器 R ，使得 $s_0 = 0$ ，因此选择器选择了输入 A 。在每个时钟周期中，只有产生指定的数据通路操作所需的控制信号才会被插入，而其他控制信号将不被插入或必要时置为不关心 (d)。NSG 模块完成由算法规定的数字通路必须实现的操作的顺序的指定。按照第 5 章中 FSM 的设计步骤可以完成这个设计。

6.3.2 微程序控制

一个微程序控制单元使用一个称为控制存储器 (CM) 的内存来存储一个称为微程序的算法的描述。这个程序由一系列的微指令组成，其中每一个都指定了一个或多个称为微操作的数据通路操作。每一条微指令也包括微程序流控制信息，它以“跳转”或“不跳转”的方

式来决定接下来将执行哪条微指令。

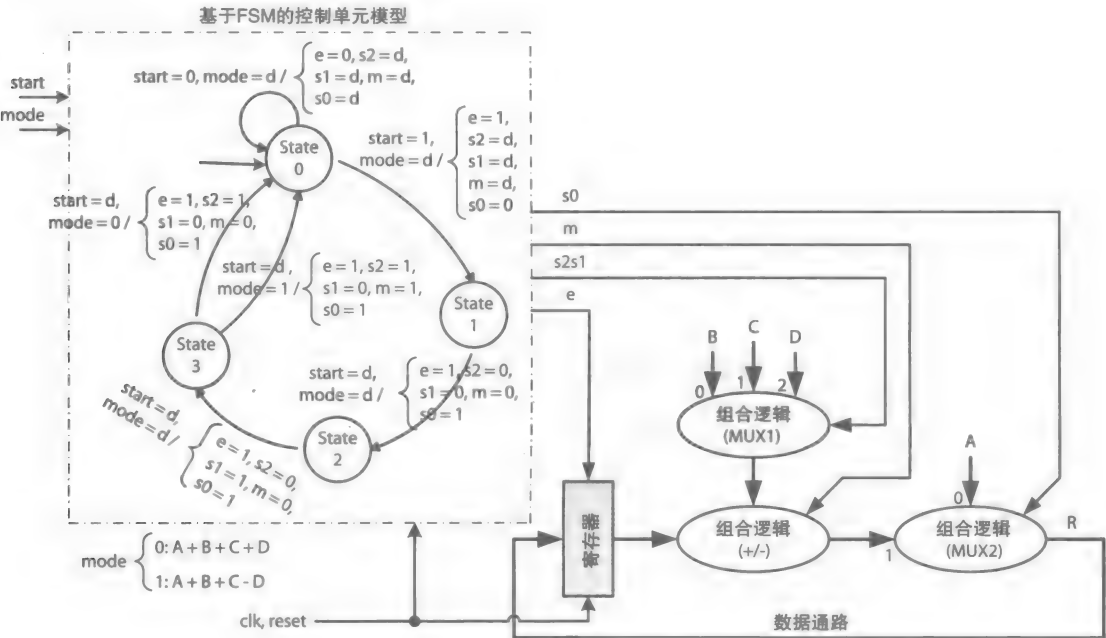


图 6-7 图 6-6 中带有实际控制信号的 FSD；d 表示不关心

227

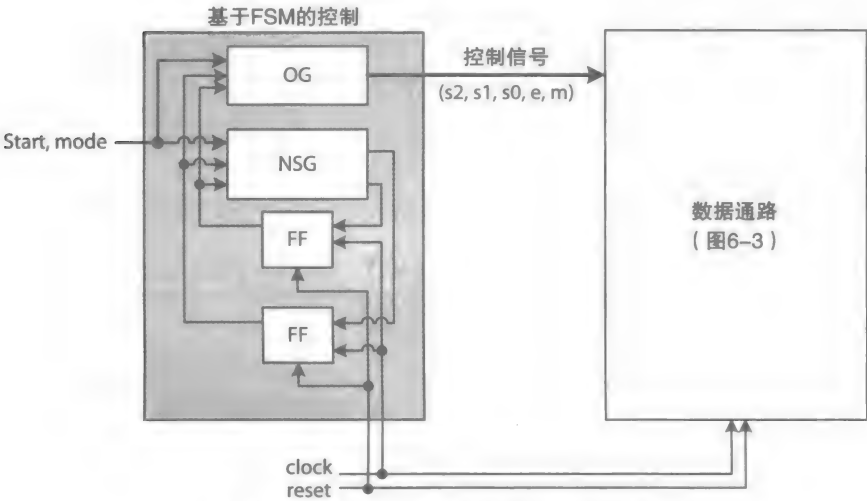


图 6-8 基于 FSM 的控制单元的详细框图

图 6-9 表示了用图 6-3 中多周期数据通路计算 $R \leftarrow A + B + C \pm D$ 的微程序。注意，微程序读起来像一个程序。它由 5 条微指令组成，其将被翻译为二进制，被称为微代码，并将被保存在 CM 的加载位置（也就是地址）0 ~ 5 处。这段微程序由三种类型的微指令组成。

地址 0 处的微指令，或者简单地说指令 0，包括条件“if start == 0”和类似 while 循环的操作，该操作每个时钟周期都会检查 start 的值，直到这个信号变为 1，然后这个控制转向了指令 1。否则，如果 start = 0，指令 0 再次执行。这个条件“if start == 0”和地址 0 产生了指令 0 的流控信息。注意，除此以外，指令 0 不包括由 RTN 所列出的任何微操作。

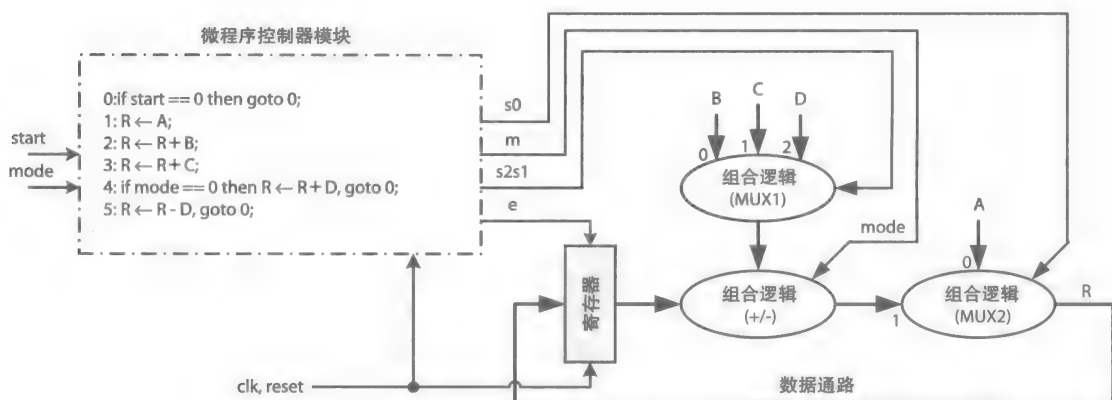


图 6-9 使用微程序控制的图 6-3 中的多周期数据通路

指令 1 ~ 3，每一个只包含一个微操作，它们不是条件性的，而是顺序执行。这三条指令的程序流控制信息是执行下一条指令。指令 4 也是条件性的。它检查是否满足 $mode = 0$ ，如果是 0，它完成一个微操作，然后跳转到指令 0，微程序的开始位置。条件“if $mode == 0$ ”和地址 0 组成了指令 4 的流控信息。否则，如果 $mode = 1$ ，指令 5（下一条指令）执行，这是一条非条件性指令，另外还表现出一个微操作（一个 RTN）。指令 5 也会表现出“go to 0”——一种非条件性的到地址 0 的跳转——其中 0 是这条指令的流控信息。

图 6-10 表示了一个微程序控制单元的详细框图。它由一个 CM 和下一个地址发生器（NAG）组成，其中后者在 CM 中产生下一条微指令的地址。它由一个叫作微程序计数器（MPC）的多功能计数器和一个 1 位的 $k-1$ 选择器组成，其中 k 是条件信号的个数（即 $start$ 和 $mode$ ）加 2；在这种情况下 $k = 4$ 。MPC 记录当前正在执行的微指令的地址。在每个时钟周期中，MPC 要么将它记录的当前指令的地址加 1，要么从 CM 中加载一个新（跳转）地址。选择器决定了哪一种功能，加 1 还是加载，然后由 MPC 进一步完成。在这个图中，MPC 被设计为当 $load = 1$ 时去加载一个跳转地址，或者当 $load = 0$ 时它的值加 1。

228

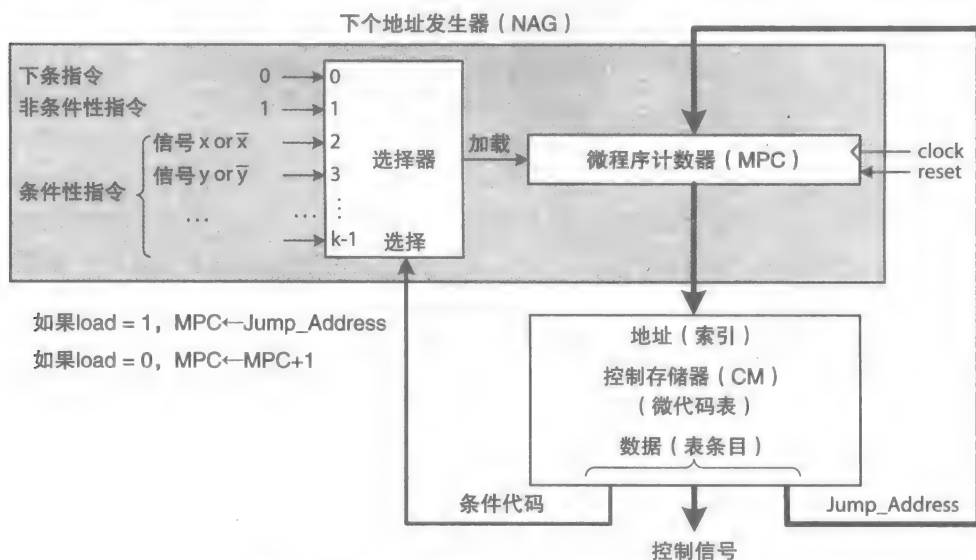


图 6-10 一个微程序控制单元的详细框图

一段微代码被组织为一张表，并存储在 CM 中。每一个表条目包括一个条件代码，一系列控制信号，可能还有一个跳转地址。表 6-2 中列出了 4 种任意指定的条件代码来实现图 6-11 中所解释并在图 6-9 中所表达的微程序控制。每一种条件代码选择出 4-1 选择器的 4 个输入中的一个来作为加载的信号值。

表 6-2 图 6-9 中微程序的条件代码

| 条件代码 (c_1c_0) | NSG (下个地址发生器) | |
|----------------------|---|---------------------------|
| | MPC | 选择器输出 |
| 00 | $MPC \leftarrow MPC + 1$ | $load = 0$ |
| 01 | $MPC \leftarrow Jump_Address$ | $load = 1$ |
| 10 | If $start == 0$ then $MPC \leftarrow Jump_Address$ else $MPC \leftarrow MPC + 1$ | $load = \overline{start}$ |
| 11 | If $mode == 0$ then $MPC \leftarrow Jump_Address$ else $MPC \leftarrow MPC + 1$ | $load = \overline{mode}$ |

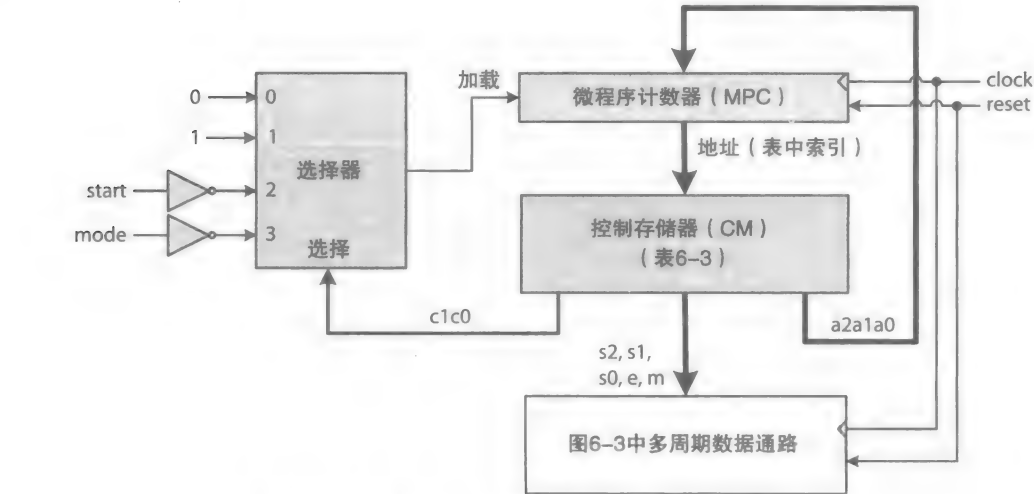


图 6-11 图 6-9 中多周期数据通路的微程序控制单元

条件代码 0 ($c_1c_0 = 00$) 被指定为这条微指令是非条件性的，比如图 6-9 中的指令 1 ~ 3。这个代码代表了“非跳转”声明（使得 $load = 0$ ），这会使指向当前微指令的 MPC 在下个时钟周期加 1。代码 1 ($c_1c_0 = 01$) 代表了一个“跳转”声明（使得 $load = 1$ ）。它被指定为这条微指令是非条件性的，比如指令 5。代码 2 ($c_1c_0 = 10$) 被指定为微指令 0。它代表了“if $start == 0$ ”条件，并通过选择器使得 $load = \overline{start}$ 。如果 $start = 0$ ，然后 $load = 1$ （跳转）；否则， $load = 0$ （不跳转）。最后，代码 3 ($c_1c_0 = 11$) 被指定为微指令 4。它代表了“if $mode == 0$ ”条件，并通过选择器使得 $load = \overline{mode}$ 。

表 6-3 列出了图 6-9 中微程序的微代码。它有 6 行，每一个 10 位的二进制是一条微指令 $\{c_1c_0, s_2, s_1, s_0, e, m, a_2a_1a_0\}$ 的代表。这个二进制的代表在最后一列中也以十六进制的形式表示了出来。

表 6-3 图 6-9 中微程序的微代码

| CM 地址 | 条件代码 | 控制信号 | | | | | 跳转地址 | 十六进制 (10-Bits) |
|-------|----------|-------|-------|-------|-----|-----|-------------|------------------|
| | C_1C_0 | S_2 | S_1 | S_0 | e | m | $a_2a_1a_0$ | |
| 0 | 10 | d | d | d | 0 | d | 000 | 200 |
| 1 | 00 | d | d | 0 | 1 | d | ddd | 010 |
| 2 | 00 | 0 | 0 | 1 | 1 | 0 | ddd | 030 |
| 3 | 00 | 0 | 1 | 1 | 1 | 0 | ddd | 070 |
| 4 | 11 | 1 | 0 | 1 | 1 | 0 | 000 | 3B0 |
| 5 | 01 | 1 | 1 | 1 | 1 | 1 | 000 | 1F8 |

d: 代表不关心, 而且存储在 CM 中时被置为 0

6.3.3 硬件控制：流水线

流水线所有阶段所需的控制信号都是一次产生, 但在恰当的时间分配给每个阶段。只完成一个简单操作的阶段不需要控制信号。图 6-12 表示了图 6-4 中带有流水线控制单元的流水线数据通路。这个数据通路实现 N 次计算, 每一次都是 $A_i + B_i + C_i \pm D_i$, 其中 $i = 0, 1, 2, \dots, N - 1$ 。信号 $mode_i$ 确定这个数据通路将要表现出的最后的算术操作是加法还是减法。这

230

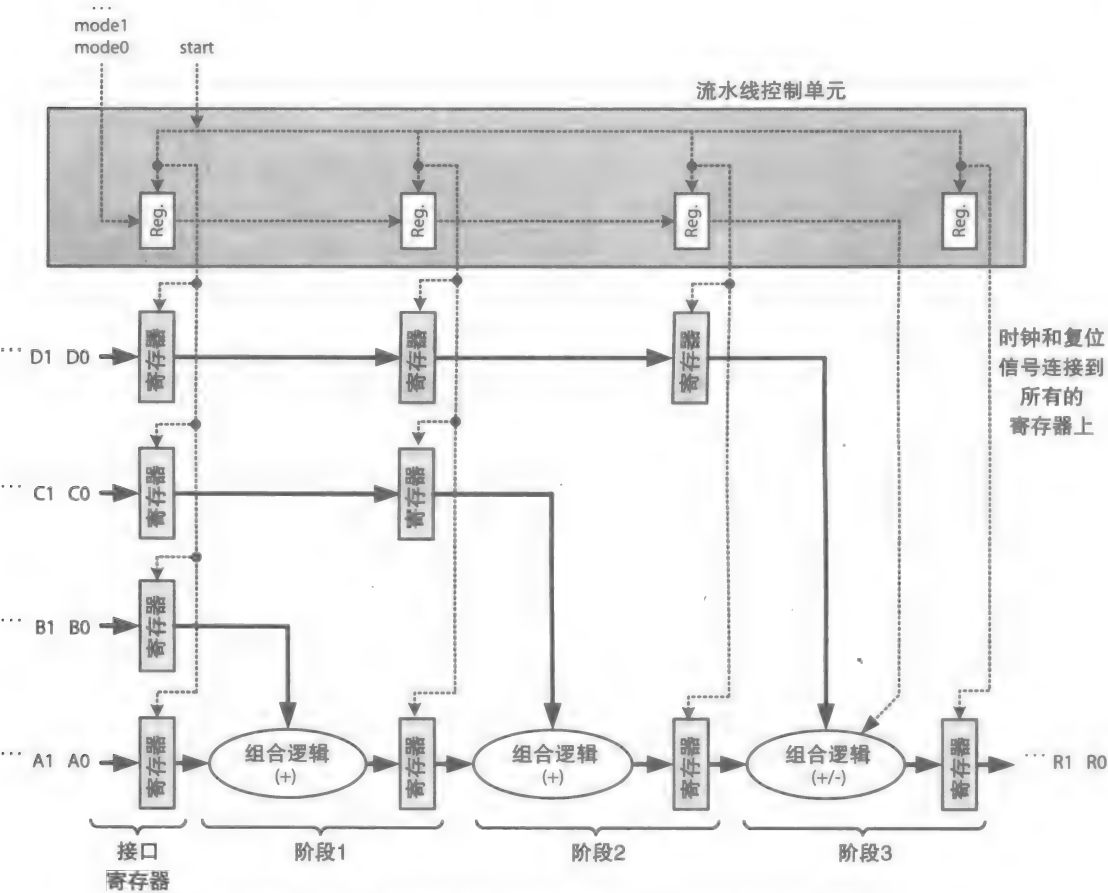


图 6-12 一个流水线控制单元和图 6-4 中的流水线数据通路

个图中所有的寄存器都被即将到达的 *start* 信号所使能。假设 *start* 将会保持逻辑 1（有效）为 $N + 4$ 个时钟周期，这是完成 N 次计算所需的时钟周期的数量（也包括接口寄存器所需的第一个时钟信号）。在 N 个时钟周期的每一个周期中，信号 $mode_i$ 和 4 个数据值 $A_i \sim D_i$ 都要送入流水线中，但是 $mode_i$ 要从一个阶段中进入下一个阶段，直到在阶段 3 中它被用来产生最终的结果，这个结果为当 $mode_i = 0$ 时的 $Y_i + D_i$ ，或当 $mode_i = 1$ 时的 $Y_i - D_i$ 。最后 4 个周期时的 $mode_i$ 的值置为不关心。注意，在这种情况下，流水线控制单元只使用一系列的寄存器，而不使用组合电路，除了 *start*，阶段 1 和阶段 2 不需要其他控制信号。

6.4 能源和功率消耗

如第 1 章所说的，随着晶体管的数量和集成芯片工作时钟频率的上升，它们也会消耗更多的功率并释放更多的热量。考虑到第 1 章中介绍过接下来也将会表示出的 CMOS 反向门电路。回忆在 CMOS 电路中，pMOS 和 nMOS 晶体管是互补的；一旦门输出稳定在逻辑 1 或逻辑 0 电平时，一个晶体管会保持打开状态，另一个保持关闭状态。

在图 6-13 中，电路中还有一个电容 C ，它的大小决定了这个门输出时为电容充电和产生逻辑 1 必要的动态电荷的数量。它被称为**动态能量**，因为门的输入和输出不是立即从逻辑 1 变为逻辑 0 或从逻辑 0 变为逻辑 1，正如 2.6 节所讨论的（第 2 章）。例如，当输入 $x = 1$ 时，pMOS 和 nMOS 晶体管分别保持关闭和打开状态（不再表示）。当 x 开始从逻辑 1 电平变为逻辑 0 电平，两个晶体管都开始切换。每个晶体管变为半开或半关状态直到 x 变为逻辑 0 电平。此时，如图中所示，pMOS 和 nMOS 都已经完全切换到了打开和关闭状态，只要 x 保持逻辑 0，它们会保持在打开和关闭状态。

[231]

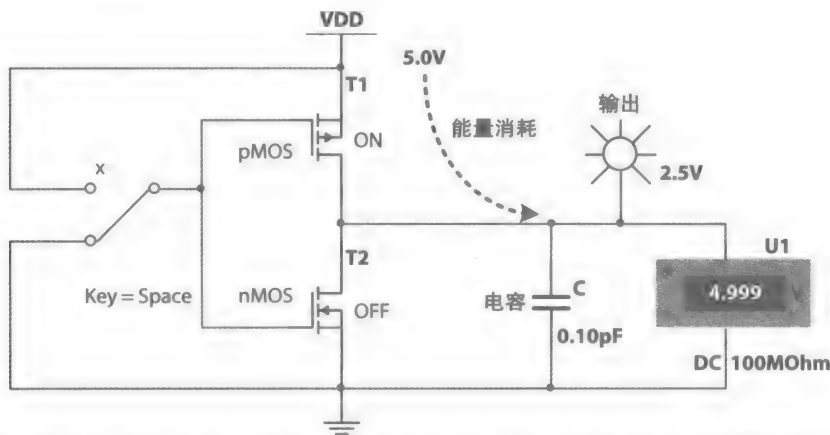


图 6-13 第 1 章中的 CMOS 非门电路描述当 x 发生 1-0 变化时电容充电，致使输出端发生 0-1 变化

在 x 发生 1-0 变化和门输出端发生 0-1 变化的晶体管转换时间，电路被称为**短路**。在这个时间里，一定量被称为**直通电流**的电流从 V_{DD} 流向地。非门发生 0-1 变化从电源输出的总能量是 CV_{DD}^2 。这些总能量中，一半作为热能消失，一半存储在电容中，正如公式（6-9）中简化形式指定的那样。

$$\text{能量存储在电容中: } E_{\text{dynamic}}^{0-1} = \int_{v=0}^{v=V_{DD}} C v \, dv = \frac{1}{2} C V_{DD}^2 \quad \text{焦耳} \quad (6-9)$$

其中“焦耳”是能量的单位。然而，非门输出端的 1-0 变化不会从能源上消耗能量。而

是存储在电容中的能量($\frac{1}{2}CV_{DD}^2$)释放到了地上,如图6-14中描述的那样。这被称为1-0变化动态能量,或者 $E_{dynamic}^{1-0} = \frac{1}{2}CV_{DD}^2$ 。

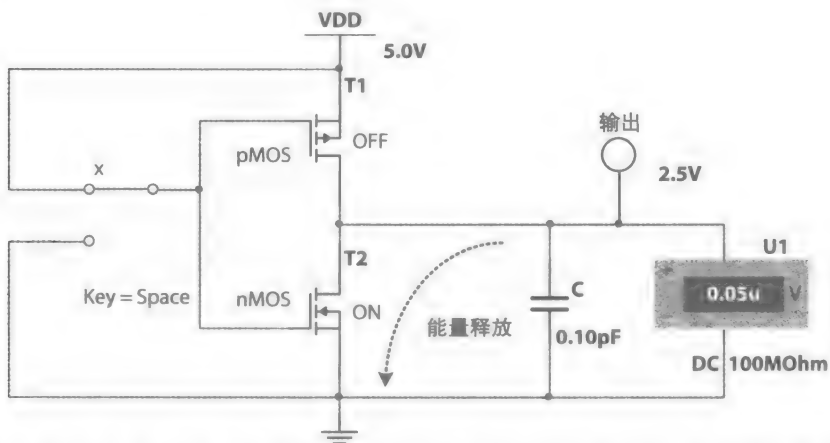


图6-14 第1章中的CMOS非门电路描述当 x 发生0-1变化时电容放电,致使输出端发生1-0变化

非门作为热能释放的动态总能量是由公式(6-10)所示的门输出端单向0-1或1-0变化引起的:

非门输出端每次0-1或1-0变化发生的热能释放的能量:

$$E_{dynamic} = \frac{1}{2}CV_{DD}^2 \quad \text{焦耳} \quad (6-10)$$

回忆在时序电路中,每个时钟周期中信号会发生0-1或1-0变化;一些信号会发生从0到1的变化,而另一些发生从1到0的变化。然后每个信号会保持它的最终逻辑0或1,直到下一个时钟周期。公式(6-11)定义了一个时序电路在一个时钟周期中消耗的总动态功率(用瓦特计)。它由电路在一秒内消耗的总动态能量(用焦耳计)确定。在公式中, τ 和 f 分别代表了时钟信号的周期和频率, C_{total} 代表了电路中总的等效电容; C_{total} 由所有门(非门、与非门等)的电容和电路中所有的连线确定。

公式(6-11)表示当时钟频率增加时,电路中的0-1和1-0变化的数量也会增加。反过来,这会增加由电路产生的热能消耗的总动态功率。这有三种方式,其中一种可能会减少一个复杂电路消耗的动态功耗:

$$P_{dynamic} = \frac{E_{dynamic}}{\tau} \quad \text{瓦特, 定义为焦耳/秒}$$

或

$$P_{dynamic} = E_{dynamic} * f \quad \text{瓦特, 其中 } f = \frac{1}{\tau} \quad (6-11)$$

$$P_{dynamic} = \frac{1}{2}C_{total}V_{DD}^2f \quad \text{瓦特}$$

- 减小总的电容, C_{total}
- 减小提供的电平, V_{DD}
- 减小时钟频率, f

然而,电容的大小或更确切地说电容的影响取决于许多参数,其中包括电路的拓扑结构

(这些门和它们的连接方式) [5]。另外, 如果电路中的电子脉冲更少一些, 消耗的动态功耗可以减少。想起这些电子脉冲在电路中是不想要的信号传输, 因此, 它们会导致不必要的电容充放电, 同时造成总动态能量作为热量的释放。因为电子脉冲发生在信号没有同时到达门的输入, 设计出带有相同上升时间和下降时间可以消除一些电子脉冲, 减少一些动态功率的使用。

除了动态功率, 电路也会消耗**静态**(待机)功率。这是当没有晶体管发生切换时电路使用的功率总量。这种消耗是因为电路输入是静态的(不发生变化), 输出是代表逻辑 1 或逻辑 0 的固定的直流电平。在这种情况下, 一定量的被称为 DC 电流 (I_{DD}) 或**漏电流**的电流会流过关闭的晶体管。公式 (6-12) 定义了电路的静态功耗。

$$P_{\text{static}} = V_{DD}I_{DD} \quad (6-12)$$

在电路中能量和功率消耗减少的同时, 比较两个复杂电路效率的首选指标是能量(比如处理器) [6]。从公式 (6-11) 中得知, 如果我们增加时钟频率, 执行任务的动态功率的消耗也会增加。然而, 动态能量的消耗保持不变(恒量)。参考一个实例, 两个处理器 A 和 B。现在假设, 在一个程序的执行中, 处理器 A 的动态功耗 P_A 比处理器 B 消耗的 P_B 大(即 $P_A > P_B$)。然而, 处理器 A 执行程序时比处理器 B 快, 即 $t_A < t_B$, 其中 t_A 和 t_B 分别代表处理器 A 和 B 执行程序的时间。在这种情况下, 也可能会发生处理器 A 比处理器 B 更节能。

假设, 对于给出的程序, P_A 比 P_B 多 20%, t_A 比 t_B 少 40%。也就是说, 处理器 A 比处理器 B 大 20% 的功率, 但执行得更快一些, 只需要处理器 B 执行程序所需时间的 60%。换句话说, $P_A = (1 + 0.2)P_B$, $t_A = (1 - 0.4)t_B$ 。因此, 从公式 (6-13) 中可知, 处理器 A 只消耗处理器 B 消耗能量的 72%。

$$\begin{aligned} P_A &= \frac{E_A}{t_A} \quad \text{从等式(6-11)的执行时间可得, 或者} \\ E_A &= P_A * t_A \\ &= (1 + 0.20)P_B * (1 - 0.30)t_B \\ &= (1.2)P_B * (0.60)t_B \\ &= (0.72)P_B * t_B \\ &= 0.72E_B \end{aligned} \quad (6-13)$$

234

甚至在程序的执行期间, 处理器 A 比处理器 B 使用更大动态功率, 但处理器 A 比处理器 B 更好一些, 因为它少消耗 28% 的总能量。处理器 B 消耗更少的动态功率, 但是因为它花费更长的时间来执行程序, 整体上, 处理器 B 比处理器 A 消耗更多的动态能量。正如第 1 章中讨论过的一样, 一个复杂的集成电路的功率和冷却的要求(即热设计功率)可以纳入其参考, 使其时钟频率可以增加, 来提高性能, 满足其冷却要求。

6.5 设计实例

在第 3 章, 一个无符号的乘法器由几个加法模块被设计成一个组合电路。作为一个时序电路, 一个多周期的乘法器可以减少硬件使用, 一个流水线式的乘法器可以提高吞吐量。表 6-4 列出了一系列的设计实例。6.5.1 节展示了无符号多周期乘法器的设计, 使用了由有限状态图设计而出的硬件控制单元。6.5.2 节展示了有符号多周期乘法器的设计, 使用了微程序控制单元。这种有符号乘法器仅用了一个加法/减法模块, 并反复地去处理两个 2 补码

数字。6.5.3 节表示了一个基本图形流水线的设计，它实现了一个二维（2-D）的 CORDIC（坐标旋转数字计算机）算法。通常，CORDIC 算法可用于实现基本的复杂函数，包括三角函数、双曲、对数、指数和平方根。表中最后两个大的时序电路的设计将在第 8 章中涉及。

表 6-4 一系列大型时序电路设计实例

| 设计实例 | 数据通路类型 | 控制类型 | |
|---------------------|--------|-------------------------------|---------|
| 无符号串行乘法器 | 多周期 | 电路控制，由有限状态图设计 | 6.5.1 节 |
| 带符号串行乘法器 | 多周期 | 微程序 | 6.5.2 节 |
| 计算机图形学：二维 CORDIC 算法 | 流水线 | 流水线式 | 6.5.3 节 |
| 单核 CPU（处理器核） | 单周期 | 电路控制，由位并行设计方法学设计（第 3 章，第 5 章） | 第 8 章 |
| 一个简单的流水线式 CPU | 流水线 | 流水线式 | 第 8 章 |

6.5.1 无符号串行乘法器

串行乘法器的优点是它仅通过一个加法器使用多周期的数据通路分布计算出两个数据的结果。[235] 每一步中，下一个加数加到之前出现加数的累加和上。如之前所讨论过的，多周期数据通路有一个缺点是比较慢，但它使用的硬件少。这一节介绍无符号乘法器的数据通路以及它基于有限状态图的控制器。这一节也将会讨论可选的硬件描述语言（HDL）设计模型，以及将会提供这个乘法器所有的 Verilog 行为建模的代码，还有仿真结果也将进行讨论。

1. 数据通路

图 6-15 描述了使用一个加法器、三个寄存器和一个模 $n+1$ 计数器设计的无符号乘法器的数据通路。A 和 B 都是 n 位的寄存器，被用来加载 n 位的被乘数 A_value 和 n 位的乘数 B_value 。P 寄存器用来加载每次两个乘法加数相加后的 $n+1$ 位的部分和（包括输出进位）。回想一下，一个加数是 A 寄存器的所有位和一个 B 寄存器位进行位运算 AND 之后的结果。这里，加数中只有不为零的位进行了加法，减少了总体的计算时间。因此，硬布线 AND 电路在第 3 章中作为组合电路被用在乘法器的设计中是不必要的。如果 $b_i = 1$ ， $addend_i = A_value$ ；否则 $addend_i = 0$ ，而且把加数加到部分和的步骤要被跳过。而且，每一个乘法步骤之后寄存器 B 将会被右移，因此它的最低有效位（LSB） b_0 用来确定下一个加数的值。计数器用来记录产生最终结果的 n 次叠加的次数。

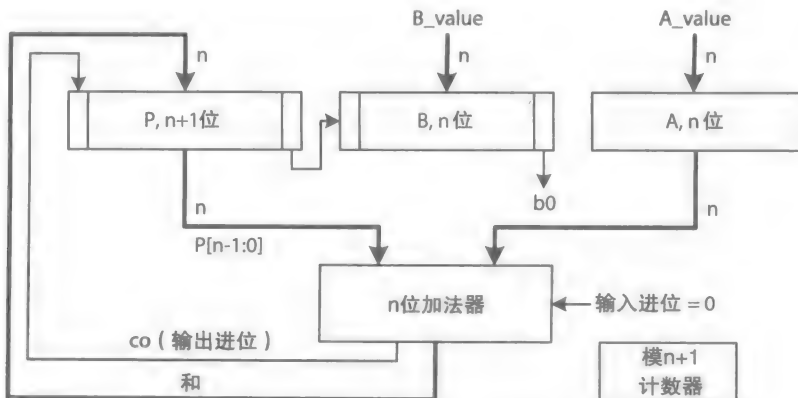


图 6-15 一个多周期无符号乘法器的数据通路

在第一个时钟周期，所有的这三个寄存器和计数器都被初始化。每次部分和产生之后，这个和都被加载到寄存器 P 中。P 和 B 寄存器然后都要右移。这样简化了算法并减少硬件数量。尤其是，这种右移会实现 1) 将当前的 b_0 放在 B 中的更高位；2) 组织好下个乘法步骤所需的部分和位；3) P 和 B 寄存器同步右移时将 P 的 LSB 加载到 B 中。最终产生的结果将被存储在 P 和 B 寄存器中。之前提到的步骤被作为乘法器算法总结如下：

236

串行无符号乘法算法

```
A ← A_value,
B ← B_value,           //Initialization, all done in
CNTR ← 0,              //one clock cycle
P ← 0;
Repeat
    If (b0 = 1) P ←      //add the next non-zero addend
    P[n-1:0] + A;        //to the current partial sum to
                        //produce a new partial sum. It is
                        //assumed that adding and
                        //loading the sum into the P takes
                        //one clock cycle

    {P, B} ← {P, B} >> 1; //right shift P and B registers
    CNTR ← CNTR + 1;      //simultaneously. This will
                        //lineup the P bits for the next
                        //cycle; and also replace b0 with
                        //the next higher bit in B. The
                        //counter is incremented to keep
                        //track of number of iterations.
                        //One clock cycle is needed to do
                        //both the shifting and the
                        //incrementing

Until CNTR < n;          //repeat n times
```

表 6-5 表示了 $A_value = 7 = (111)_2$ 和 $B_value = 5 = (101)_2$ 时的无符号乘法器算法的每一步说明。三步之后，放在 $\{P[2:0], B\}$ 中的 6 位结果是 $(100, 011)_2$ ，或者说十进制中的 35。

表 6-5 无符号数 $A_value = (111)_2$ 和 $B_value = (101)_2$ 相乘的分布说明

| 模 3 计数器 | P[3], P[2:0] | B | A | 备 注 |
|---------|--------------|-----|-----|---|
| 0 | 0, 000 | 101 | 111 | 初始化 |
| | 0, 111 | 101 | 111 | $b_0 = 1$ ，因此 $P \leftarrow P[2:0] + A$ |
| 1 | 0, 011 | 110 | 111 | $\{P, B\}$ 右移并填充 0 |
| | 0, 011 | 110 | 111 | $b_0 = 0$ ，因此 P 保持不变 |
| 2 | 0, 001 | 111 | 111 | $\{P, B\}$ 右移并填充 0 |
| | 1, 000 | 111 | 111 | $b_0 = 1$ ，因此 $P \leftarrow P[2:0] + A$ |
| 3 | 0, 100 | 011 | 111 | $\{P, B\}$ 右移并填充 0 |

2. 控制单元设计：FSD

使用 HDL 和 / 或原理图设计工具去设计一个大的时序电路有多种方式。下面概括了所有类型的数据通路和控制器的三种通用的设计实践：

I 全结构式——这种设计将会用到相互关联的模块的层次结构，所有层次结构中的叶子模块将使用布尔表达式（即使用 assign 语句）或电路（即使用基本门）进行建模。这些模块然后将通过显式声明的控制信号进行内部互连。这种选择不推荐在非常大的设计中使用。另外，可以使用原理图设计工具。

Ⅱ 混合式——这种设计将同时使用结构建模和行为建模。在这种情况下，分层模块将为叶子模块使用行为建模（即 `always` 块），然后使用显式声明的控制信号进行内部互连。也可以使用带 HDL 接口的原理图设计工具。

Ⅲ 全行为式——这种设计描述带有由 RTN 所表示数据通路操作的有限状态图的电路模块的行为。这种设计不需要明确的 RTN——相关的控制信号。

首先我们先说明无符号乘法电路的设计要求，先由图 6-15 中操作每一个寄存器和计数器所需的明确声明的控制信号开始。表 6-6 列出了一系列每一个寄存器 A、B、P 和计数器（CNTR）所必须实现的功能。A 是一个单功能并行加载寄存器；B 是一个双功能并行加载和右移寄存器；P 是一个三功能的并行加载、右移和同步清零寄存器。CNTR 是一个双功能的向上计数和同步清零计数器。

表 6-6 RTN 表示的寄存器和计数器功能

| 功 能 | 含 义 |
|------------------------------------|--------------|
| $A \leftarrow A_value;$ | 并行加载 |
| $B \leftarrow B_value;$ | 并行加载 |
| $B \leftarrow \{P[0], B[n-1:1]\};$ | 右移，左端输入 (Li) |
| $P \leftarrow 0;$ | 同步初始化为 0 |
| $P \leftarrow \{co, sum\};$ | 并行加载 |
| $P \leftarrow \{0, P[n:1]\};$ | 右移并填充零 |
| $CNTR \leftarrow 0;$ | 同步初始化为 0 |
| $CNTR \leftarrow CNTR + 1;$ | 增量 |

对于所有的结构式（选择 I）或混合式（选择 II）设计，所有的寄存器和计数器必须被指定到一系列的控制信号上，即图 6-16 中所示。一个简单的组合电路（CC）用于将一个 CNTR 产生的多位的输出转换为控制单元所用的单位的标志信号。如果 $count = n$, $flag = 1$ ；否则， $flag = 0$ 。表 6-7 列出了明确的控制信号的值和每一个相关的数据通路操作。假设这些寄存器和计数器由不带使能信号的触发器设计实现。

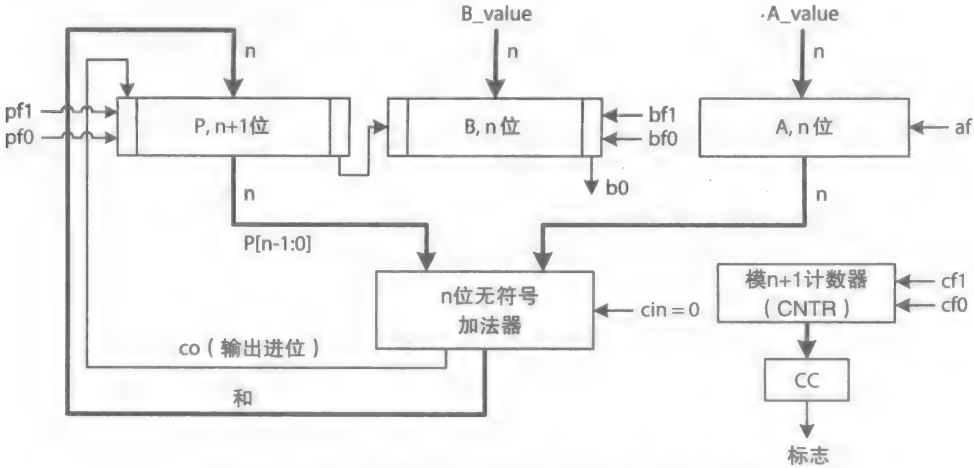


图 6-16 带有控制信号的无符号乘法器的数据通路

表 6-7 HDL 结构建模的图 6-16 数据通路的控制信号

| 控制信号 | | 描 述 |
|------|-----|-----------------------|
| af | | 寄存器 A 的单控制信号，使能或禁用寄存器 |
| 1 | | 使能寄存器 A 进行并行加载 |
| 0 | | 禁用寄存器 A |
| bf1 | bf0 | 寄存器 B 的控制信号 |
| 0 | 0 | 禁用寄存器 B |
| 0 | 1 | 使能寄存器 B 进行右移 |
| 1 | 0 | 使能寄存器 B 进行右移 |
| 1 | 1 | 未使用 |
| pf1 | pf0 | 寄存器 P 的控制信号 |
| 0 | 0 | 禁用寄存器 P |
| 0 | 1 | 使能寄存器 P 进行并行加载 |
| 1 | 0 | 使能寄存器 P 进行右移 |
| 1 | 1 | 使能并同步清零寄存器 P |
| cf1 | cf0 | |
| 0 | 0 | 禁用计数器 |
| 0 | 1 | 使能计数器进行增量计数 |
| 1 | 0 | 未使用 |
| 1 | 1 | 使能并同步清零计数器 |

图 6-17 表示了无符号乘法电路的详细框图。外部的触发信号 *start* 由一个同步的触发器 (FF1) 输出而得，由它启动这个乘法器的控制单元。信号 *done* (Mealy) 在计算结束的时候被插入，并被作为 *done_moore* 保存在另一个触发器 (FF2) 中。如果 *_reset* = 0 或 *done* = 1，FF1 被复位，如果 *_reset* = 0 或 *start_asyn* = 1，FF2 被复位。

238

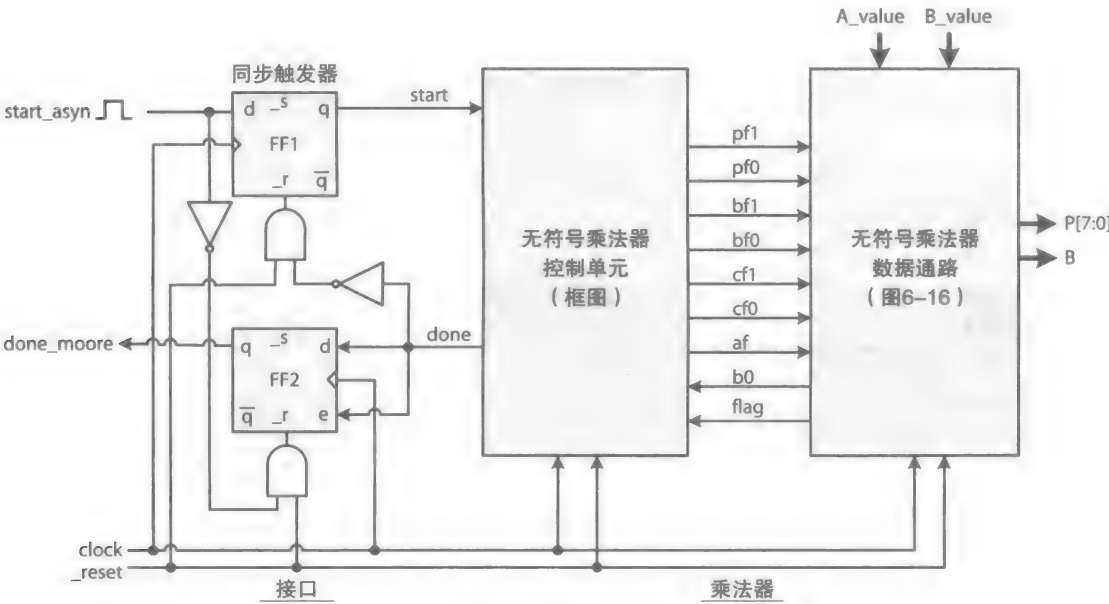


图 6-17 无符号乘法器框图、控制信号和接口信号

图 6-18 表示了乘法器控制单元的有限状态图，并带有由 RTN 表示的数据通路操作。有

限状态图由标识为 Idle、Check 和 Add 的三种状态组成。如有限状态图所示，复位后，控制单元初始化到 Idle 状态。一旦进入 Idle 状态，控制单元便监视 *start* 信号，直到信号变为 1，并开始触发乘 *A_value* 和 *B_value* 的控制单元，即按之前讨论过的无符号乘法算法进行。

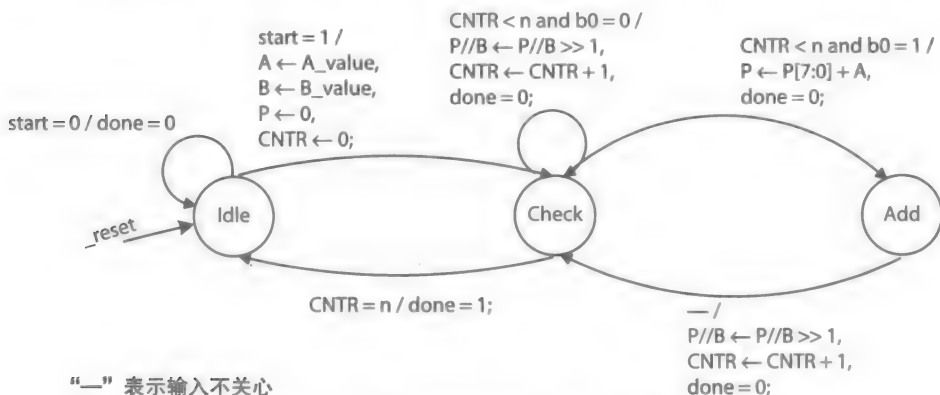


图 6-18 无符号乘法器控制器的有限状态图；只包含了非零的加数

3. HDL 模型

乘法器的全结构式的和混合式的设计都被推迟到了练习部分。然而，使用明确声明的控制信号的设计实例在第 6.5.2 节中。接下来是图 6-17 中无符号乘法器和它的接口模块的全行为式（选择Ⅲ）的 HDL 代码。HDL 代码建模的图 6-18 中有限状态图带有未明确声明的数据通路控制信号。特别地，代码中描述了一个带有 NSG、OG 和一系列触发器的 FSM。OG 负责产生由 RTN 隐式指定的数据通路控制信号。因此，全行为式的 OG 将使用 RTN 建模出带有隐式控制信号的乘法器数据通路。

例 6-2 下面描述了图 6-17 中无符号乘法器和它的接口模块的 Verilog 行为建模。描述中使用了未明确声明的数据通路控制信号。

解：这个乘法器完全是由图 6-18 中通过 RTN 给出的数据通路操作的有限状态图的指定性描述。

```

module interface_unit(
    input clock, _reset, start_asyn, done,
    output reg start, done_moore
);
//----- synchronization flip-flop -----
always@(posedge clock or negedge _reset or posedge done)
begin
    if(_reset == 0 || done == 1)
        start <= 1'b0;
    else
        start <= start_asyn;
end
//----- Convert a Mealy output to a Moore output -----
always@(posedge clock or negedge _reset or posedge start_asyn)
begin
    if(_reset == 0 || start_asyn == 1)
        done_moore = 1'b0;
    else
        if(done == 1)
            done_moore <= done;
end
end
  
```



```

endmodule
//----- Unsigned Multiplier -----
//A Mealy controller FSM
module umult(
    input clock, _reset, start,
    input [7:0] a_value, b_value,
    output [15:0] result,
    output reg done
);
reg [8:0] p;
reg [7:0] a, b;
reg [3:0] cntr; //mod-16 counter
reg co;
reg [7:0] sum;
reg [1:0] current_state, next_state;
assign result = {p[7:0], b};
//----- The states -----
parameter Idle = 2'b00,
           Check = 2'b01,
           Add = 2'b10;

/*----- Output Generator (OG) -----
The OG module defines the data path and also implicitly defines
the data path control signals using a behavior description*/
always@(posedge clock or negedge _reset)
begin
    if(!_reset)
        begin
            p <= 0;
            cntr <= 0;
        end
    else
        case(current_state)
            Idle: if(start == 1)
                begin // initialize
                    a <= a_value;
                    b <= b_value;
                    p <= 0;
                    cntr <= 0;
                end
            Check: begin
                if (cntr < 8)
                    if(b[0] == 1)
                        p <= {co, sum}; //or p <= p[7:0] + a; //without delay
                    else
                        begin
                            {p, b} <= {p, b} >> 1;
                            cntr <= cntr + 1;
                        end
                    end
                Add:begin
                    {p, b} <= {p, b} >> 1;
                    cntr <= cntr + 1;
                end
            endcase
        end
    always@(current_state or cntr or start or b[0])
begin
    case(current_state)
        Idle:begin
            done = 0;

```

```

        if(start == 1)
            next_state = Check;
        else
            next_state = Idle;
        end
    Check: if(cntnr < 8)
        begin
            done = 0;
            if(b[0] == 0)
                next_state = Check;
            else
                next_state = Add;
            end
        else
            begin
                done = 1;
                next_state = Idle;
            end
        Add: begin
                done = 0;
                next_state = Check;
            end
        default:begin
            done = 0;
            next_state = Idle;
        end
    endcase
end
//----- The flip-flops -----
always@(posedge clock or negedge _reset) //state transitions
begin
    if (!_reset)
        current_state <= Idle;
    else
        current_state <= next_state;
    end

//adder with delay
always@(p or a)
begin
    //assume 10ns delay for the adder
    #10 {co, sum} = p[7:0] + a;
end
endmodule

```

4. 仿真

乘法器和它的接口模块通过 Altera Quartus II 和 Altera ModelSim 10.1b 进行了综合和仿真。这个工具还提供了一个 state machine viewer 的验证功能，它可以通过给出的 Verilog 描述重建出有限状态图。图 6-19 表示了重建的例 6-2 中 Verilog 所描述乘法器控制单元的有限状态图。

例 6-3 是带有两个测试向量的测试平台。图 6-20 中表示了相关的仿真时序图。注意，由于该算法跳过为零的加法，乘法器计算 $B_value = 8'h7F$ 乘 $A_value = 8'h03$ 比计算 $B_value = 8'h03$ 乘 $A_value = 8'h7F$ 花费更长的时间，因为 $B_value = 8'h7F$ 中有 7 个 1，而 $B_value = 8'h03$ 中只有两个 1。另外，乘法器可以被设计成带有一个比较器，如果 A_value

中 1 的个数更少，就交换操作数 A_value 和 B_value 的值。

243

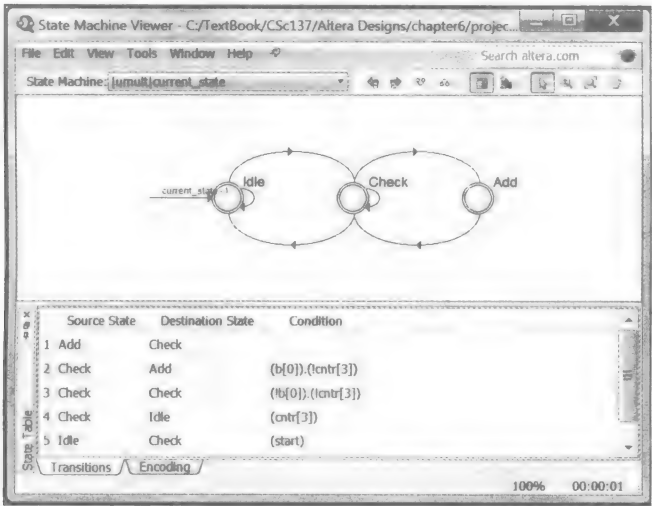


图 6-19 例 6-2 中乘法器 Verilog 描述重建的有限状态图

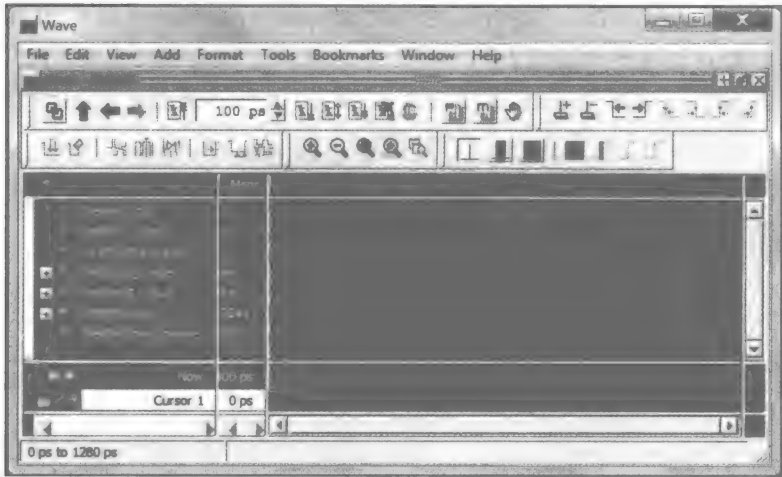


图 6-20 图 6-17 中乘法器的仿真输出，使用了例 6-3 中的测试台

例 6-3 这里描述了带有两个测试用例 $8'h03 \times 8'b7F$ 和 $7'h7F \times 8'h03$ 的 HDL 测试平台。

```
module tester();
  reg clock, _reset, start_async;
  reg [7:0] a_value, b_value;
  wire [15:0] result;
  wire done, done_moore, start;

  interface u1(clock, _reset, start_async, done, start, done_moore);
    umult u2(clock, _reset, start, a_value, b_value, result, done);
  endinterface

  initial begin
    clock = 1;
    #10 forever #10 clock = ~clock; //20ns clock period
  end
```

```

initial begin
  _reset = 0;
  start_asyn = 0;
  #15 _reset = 1;
  #10 start_asyn = 1; a_value = 8'h03; b_value = 8'h7F;
  #40 start_asyn = 0;

  #400 start_asyn = 1; a_value = 8'h7F; b_value = 8'h03;
  #40 start_asyn = 0;

  #1000 $finish;
end

initial
  $monitor($stime,, _reset,, start_asyn,, clock,,, result,, done_
moore);
endmodule

```

244

6.5.2 带符号串行乘法器

2 的补码乘法算法，被普遍称为 Booth 算法，使用加法和减法去乘两个 2 的补码的正数或负数。它的数据通路和之前讨论过的带三个寄存器 A、B 和 P 的无符号乘法相似。A 的 2 的补码被乘数 A_value 和一个二进制补码的乘数 B_value 分别被加载到 A 和 B 寄存器中，最终的结果要从 P 和 B 寄存器中被读出。

在 Booth 算法中，一系列的 1，例如三个 1 或者 $(111)_2$ ，被表示为 $(100\bar{1})_2$ ；其中 $\bar{1}$ 在这里表示 -1。 $(111)_2$ 和 $(100\bar{1})_2$ 都表示十进制中的 7； $(111)_2 = 4 + 2 + 1$ 是 7，所以 $(100\bar{1})_2 = 8 - 1$ 。这种解释是由仅两部分和的计算取代了需要典型 A_value 乘 7 的三部分和的计算：序列的开始是减法，序列的结束是加法。中间的 1 将被翻译为 0，并被跳过。

这是通过检查值 $\{B_value, 0\}$ （即 B_value 与 0 拼接）实现的，从它的 LSB 开始，一次两位但是重叠，使用了表 6-8 中指定的方式去用 B_value 乘 A_value 。

表 6-8 Booth 乘法器的位解释

| b_0 | b_{-1} | 含 义 |
|------------------|----------|--|
| 0 | 0 | 跳过：0 的序列； $\{P, B\} \leftarrow \{P, B\} \ggg 1$. |
| 0 | 1 | 加：意味 1 序列的结束； $P \leftarrow P + A$. |
| 1 | 0 | 减：意味 1 序列的开始； $P \leftarrow P - A$. |
| 1 | 1 | 跳过：1 序列被解释为 0； $\{P, B\} \leftarrow \{P, B\} \ggg 1$. |
| >>> 表示带符号扩展的算术右移 | | |

1. 数据通路

245

图 6-21 表示了 Booth 乘法器的数据通路。 x 和 b_1 信号被控制单元使用。如果 $x = 0$ ，寄存器 P 和 B（表示为 $\{P, B\}$ ）同时算术右移，保存 P 的 LSB 在 B 中。如果 $x = 1$ ， $b_1 = 0$ ， A_value 的值加到 P 的值上；否则，就从 P 的值中减去 A_value 。无符号乘法器中的组合电路 (CC) 将计数器的输出转换为信号 $flag$ 。如果 $count = n$ ， $flag = 1$ ，否则， $flag = 0$ 。

除了 $n + 1$ 位的 B 寄存器，A 和 P 寄存器也都是 $n + 1$ 位的，所以乘法器电路可处理的最大数量级是 n 位的 2 的补码的负数。例如， $A_value = -8$ 或者说 $(1000)_{2s}$ 是最小的 4 位的 2 的补码负数，P 的值是 0（即 $P_value = 0$ ）。 $P_value - A_value$ 的值应该是 +8，现在，这个值将被错误地翻译为 4 位的 $(1000)_{2s}$ ，这是 4 位的 2 的补码 -8 的表示。因此，让 A

和 P 都用 5 位的寄存器就能解决这个问题。5 位的 -8 表示为 $(11000)_{2s}$ ，放在 A 寄存器中， $(00000)_{2s} - (11000)_{2s}$ 表示为 $(01000)_{2s} = +8$ ，正确地存在 P 寄存器中。

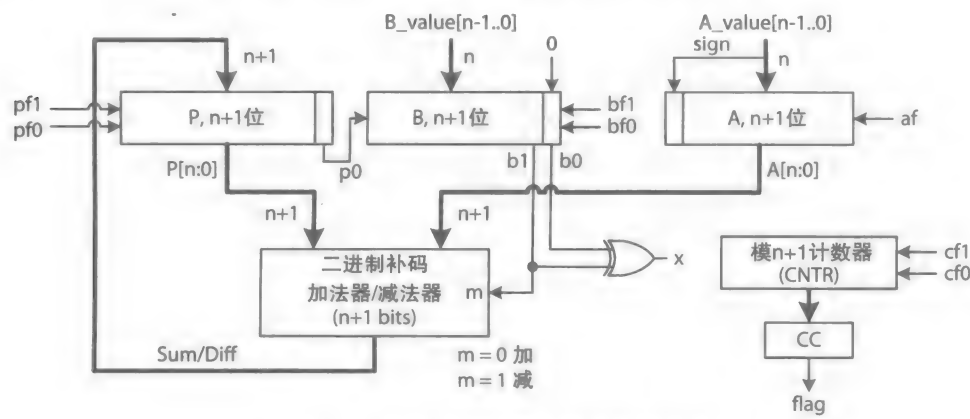


图 6-21 串行 Booth 乘法器的数据通路

B 寄存器中存储 $n + 1$ 位的值 $\{B_value, 0\}$ 。假设 4 位的 $B_value = (1111)_{2s} = -1$ 。注意 $A_value \times B_value$ 或者说 $A_value \times -1$ 的结果应该是 $-A_value$ ，其中 A_value 是一个任意的 4 位的 2 的补码数。起初，如在数据通路时介绍的那样， B_value 将存在 5 位的 B 寄存器中表示为 $(11110)_{2s}$ ，其中有它的两位 LSB 位 $b_1b_0 = (10)_2$ 。使用表 6-8 中的规则，如果 $b_1b_0 = (10)_2$ ，那么初值为 0 的 P 寄存器的值将变为 $0 - A_value = -A_value$ 。然后， $\{P, B\}$ 将进行算术右移，并复制 P 的符号位，符号位现在为 1。因为 B 寄存器剩余的位都是 1， $\{P, B\}$ 将算术右移 4 次，每次都重复 P 的符号位。这将在 $\{P_{n-1..0}, B_{n..1}\}$ 中产生最终的正确结果 $-A_value$ ，是一个 $2n$ 位的 2 的补码负数。表 6-9 描述了使用图 6-21 中数据通路并且 $n = 4$ 时的 -8×-5 的过程。这个结果是 $40 = (0010, 1000)_{2s} = 8'h28$ 。

246

表 6-9 4 位的 Booth 乘法示例： $A_value = -8 = (1000)_{2s}$ ， $B_value = -5 = (1011)_{2s}$

| Mod-5 CNTR | P | B | A | 备 注 |
|------------|-------|-----------------|---------|----------------------------------|
| 0 | 00000 | 1011 <u>1</u> 0 | 1, 1000 | 初始化 |
| 1 | | | | $b_1b_0 = 10$ ，减、加载和移位 |
| | 01000 | 1011, 0 | 1, 1000 | $P \leftarrow P - A$; |
| | 00100 | 0101 <u>1</u> 1 | 1, 1000 | $\{P, B\} \ggg 1$ 且 CNTR++; |
| 2 | | | | $b_1b_0 = 11$ ，移位 |
| | | 0010 <u>1</u> 1 | 1, 1000 | $\{P, B\} \ggg 1$ 且 CNTR++; |
| 3 | | | | $b_1b_0 = 01$ ，加，load, and shift |
| | 11010 | 0010, 1 | 1, 1000 | $P \leftarrow P + A$; |
| | 11101 | 0001 <u>1</u> 0 | 1, 1000 | $\{P, B\} \ggg 1$ 且 CNTR++; |
| 4 | | | | $b_1b_0 = 10$ ，减、加载和移位 |
| | 00101 | 0001, 0 | 1, 1000 | $P \leftarrow P - A$; |
| | 00010 | 1000, 0 | 1, 1000 | $\{P, B\} \ggg 1$ 且 CNTR++; |

2. 乘法器算法：微程序

假设图 6-17 中接口模块也用来产生两个接口信号 *start* 和 *done*，表 6-10 列出了控制图 6-21 中带符号乘法器的微程序。

表 6-10 控制图 6-21 中带符号乘法器数据通路的微程序

| 地址 | 微指令 | 备注 |
|----|---|--|
| 0 | If start == 0 go to 0; | 等待开始 |
| 1 | $P \leftarrow 0, A \leftarrow A_value, B \leftarrow B_value, CNTR \leftarrow 0$; | 初始化 |
| 2 | If CNTR == n go to 6; | 当 n 次迭代完成时结束 |
| 3 | If x == 0 go to 5 | 如果 $b_1b_0 = 00$ 或 11 时跳过 |
| 4 | $P \leftarrow P \pm A$; | 计算并加载部分和, 如果 $b_1 = 0$ 用加法, 如果 $b_1 = 1$ 用减法 |
| 5 | {P, B} >>> 1, CNTR \leftarrow CNTR + 1, go to 2; | 更新 P、B 和 CNTR, 然后重复 |
| 6 | done = 1, go to 0; | 结束, 转到 0 |

247

3. 控制单元设计：微程序

图 6-22 表示了带符号乘法器的微程序控制单元。条件 “if start == 0”、“if CNTR == n” 和 “if x == 0” 分别对应条件代码 2、3 和 4。微程序计数器（MPC）在 load = 1 时加载跳变地址（ $a_2a_1a_0$ ），或者 load = 0 时让它的值增加。表 6-11 列出了微程序的微代码。

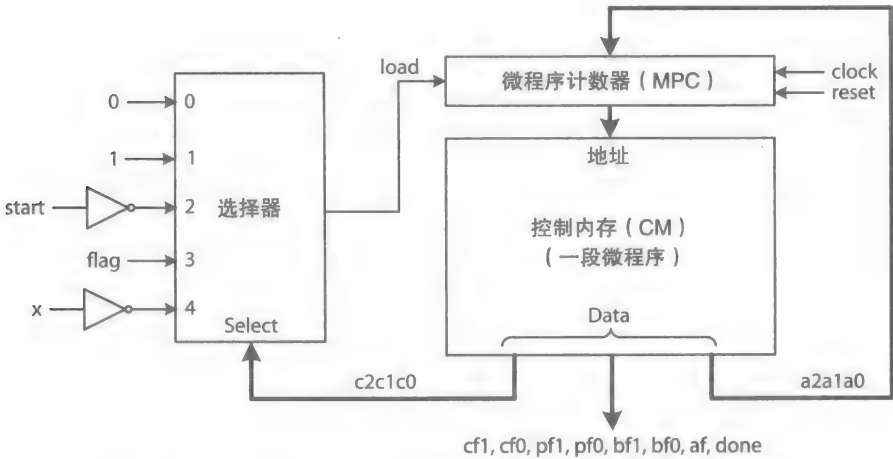


图 6-22 图 6-21 中 Booth 乘法器数据通路的微程序控制单元

表 6-11 表 6-10 中微程序的微代码

| CM 地址 | 条件码 | 控制信号 | | | | | 跳转地址 | 十六进制 (14 bits) |
|-------|-------------|------------|------------|------------|----|------|-------------|---------------------|
| | $C_2C_1C_0$ | cf_1cf_0 | pf_1pf_0 | bf_1bf_0 | af | done | $a_2a_1a_0$ | |
| 0 | 010 | 00 | 00 | 00 | 0 | 0 | 000 | 1000 |
| 1 | 000 | 11 | 11 | 01 | 1 | 0 | ddd | 07B0 |
| 2 | 011 | 00 | 00 | 00 | 0 | 0 | 110 | 1806 |
| 3 | 100 | 00 | 00 | 00 | 0 | 0 | 101 | 2005 |
| 4 | 000 | 00 | 01 | 0 | 0 | 0 | ddd | 0080 |
| 5 | 001 | 01 | 10 | 10 | 0 | 0 | 010 | 0B42 |
| 6 | 001 | 00 | 00 | 00 | 0 | 1 | 000 | 0808 |

d 代表了不关心，产生十六进制数值时用 0 替代

4. HDL 模型

例 6-4 描述了图 6-22 中的微程序控制单元和图 6-21 中带有明确指定的控制信号的数据

通路。数据通路的描述包括了 CM 的初始化。

248

例 6-4 下面表示了使用 HDL 结构和行为描述的 2 的补码 Booth 乘法器的 Verilog 建模。

解：假设使用图 6-17 中的接口去连接控制单元。

```

module smult(
    input clock, _reset, start,
    input [7:0] a_value, b_value,
    output [15:0] result,
    output done
);
wire flag, x;
wire [13:0] control;

controller u2(clock, _reset, start, flag, x, control, done);
data path u3(clock, _reset, a_value, b_value, control, flag, x,
result);

endmodule

module controller(
    input clock, _reset, start, flag, x,
    output reg[6:0] control,
    output reg done
);

reg [2:0] mpc; //control memory
(* ramstyle = "M512" *) reg [13:0] cm[0:7]; //control memory, using
//a built-in memory

//reg [0:7][13:0] cm;
reg [2:0] ccode; //branch type
reg [2:0] jump_address; //branch address
reg load;

//-----Initialize the CM,-----
initial begin
    cm[0] = 14'h1000; //wait for start = 1
    cm[1] = 14'h07B0; //initialize
    cm[2] = 14'h1806; //if flag = 1 then goto 6
    cm[3] = 14'h2005; //if x = 0 then goto 5
    cm[4] = 14'h0080; //p <- sum_diff
    cm[5] = 14'h0B42; //p//b <= p//b >>> 1, cntr++, goto 2
    cm[6] = 14'h0808; //done = 1, goto 0
end

//----- MUX -----
always@(*)
begin
    case(ccode)
        0: load = 0; //next instruction
        1: load = 1; //unconditional jump
        2: if (start == 0) //conditional jump if start = 0
            load = 1;
        else
            load = 0;
        3: if (flag == 1) //conditional jump if done
            load = 1;
        else
            load = 0;
        4: if (x == 0) //conditional jump if b[1]b[0] = 00 or 11
            load = 1;
        else
    
```

249

```

        load = 0;
    default: load = 1;
    endcase
end

//----- MPC -----
always@(posedge clock or negedge _reset)
begin
    if (_reset == 0)
        mpc <= 3'b000;
    else
        if(load == 0)
            mpc <= mpc + 1;
        else
            mpc <= jump_address;
    end
end

//----- CM -----
always@(*)
begin
    ccode = cm[mpc][13:11];
    control = cm[mpc][10:4];
    done = cm[mpc][3];
    jump_address = cm[mpc][2:0];
end
endmodule

module data_path(
    input clock, _reset,
    input [7:0] a_value, b_value,
    input [6:0] control,
    output reg flag,
    output x,
    output [15:0] result
);

reg [8:0] a, b;
reg [8:0] p;
reg [3:0] cntnr; //mod-16 counter
reg [8:0] sum_diff;

wire af = control[0];
wire [1:0] CF = control[6:5],
           PF = control[4:3],
           BF = control[2:1];
assign result = {p[7:0], b[8:1]};
assign x = b[1] ^ b[0];

always@(posedge clock or negedge _reset) //registers and cntnr
begin
    if (_reset == 0)
    begin
        a <= 0;
        b <= 0;
        p <= 0;
        cntnr <= 0;
    end
    else
    begin
        if (af == 1)
            a <= {a_value[7], a_value}; //a[8] is set to the sign bit
        (a[7])
    end
end

```



```

    case(BF)
        2'b01: b <= {b_value, 1'b0};
        2'b10: b <= {p[0], b[8:1]}; //shift right with left input
        2'b11: b <= 9'h000; //clear
        default: b <= b;
    endcase
    case(PF)
        2'b01: p <= sum_diff; //load
        2'b10: p <= {p[8], p[8:1]}; //arithmetic right shift
        2'b11: p <= 9'h000; //clear
        default: p <= p;
    endcase
    case(CF)
        2'b01: cntr <= cntr + 1; //increment
        2'b10: cntr <= cntr; //not used, retain
        2'b11: cntr <= 3'b000; //clear
        default: cntr <= cntr;
    endcase
end

always@(*)
begin
    if (b[1] == 1'b0)
        sum_diff = p + a;
    else
        sum_diff = p - a;
end

always@(*) //condition flag
begin
    if (cntr < 8)
        flag = 1'b0;
    else
        flag = 1'b1;
end

end
endmodule

module interface_unit(
    input clock, _reset, start_asyn, done,
    output reg start, done_moore
);
//synchronization flip-flop
always@(posedge clock or negedge _reset or posedge done)
begin
    if(_reset == 0 || done == 1)
        start <= 1'b0;
    else
        start <= start_asyn;
end

//Moore output
always@(posedge clock or negedge _reset or posedge start_asyn)
begin
    if(_reset == 0 || start_asyn == 1)
        done_moore = 1'b0;
    else
        if(done == 1)
            done_moore <= done;
end
endmodule

```

5. 仿真

例 6-5 描述了带符号乘法器的测试平台。图 6-23 表示了解释 -8 乘 -5 的仿真时序图。结果是十进制的 40。

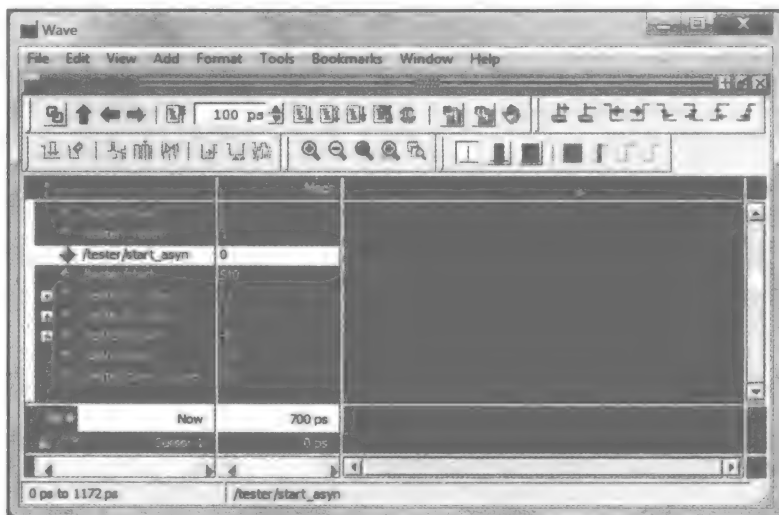


图 6-23 -5 乘 -8 的仿真输出

例 6-5 下面描述了计算 -8×-5 或者十六进制的 $0xF8 \times 0xFB$ 或者 Verilog 中的 $8'hF8 \times 8'hFB$ 的测试平台。假设使用图 6-17 中的接口模块去连接控制单元。

```

module tester();
  reg clock, _reset, start_async;
  reg [7:0] a_value, b_value;
  wire [15:0] result;
  wire done, done_moore, start, flag, x;
  wire [6:0] control;

  interface_unit u1(clock, _reset, start_async, done, start,
    done_moore);
  smult u2(clock, _reset, start, a_value, b_value, result,
    done);

  initial begin
    clock = 1;
    _reset = 0;
  end

  always #10 clock = ~clock; //20ns clock period

  initial begin
    start_async = 0;
    #15 _reset = 1;
    #10 start_async = 1; a_value = 8'hF8; b_value = 8'hFB;
    #40 start_async = 0;

    #1000 $finish;
  end

  initial begin
    $monitor($stime, _reset, start_async, clock, result, done_moore);
  end
endmodule

```

6.5.3 计算机图形学：旋转

在计算机图形学中，一个虚拟对象被定义为笛卡儿坐标中的几个点。例如，图 6-24 中所示，一个二维的虚拟对象“房子”被它的 5 个点所表示，分别被记为 $x-y$ 坐标系统中的 a 到 e 。每个点都被视为从原点 $(0, 0)$ 到坐标 (X, Y) 的一个向量。这些向量 a 到 e 的坐标分别是 $(10, 10)$, $(10, 20)$, $(15, 30)$, $(20, 20)$ 和 $(20, 10)$ 。

253

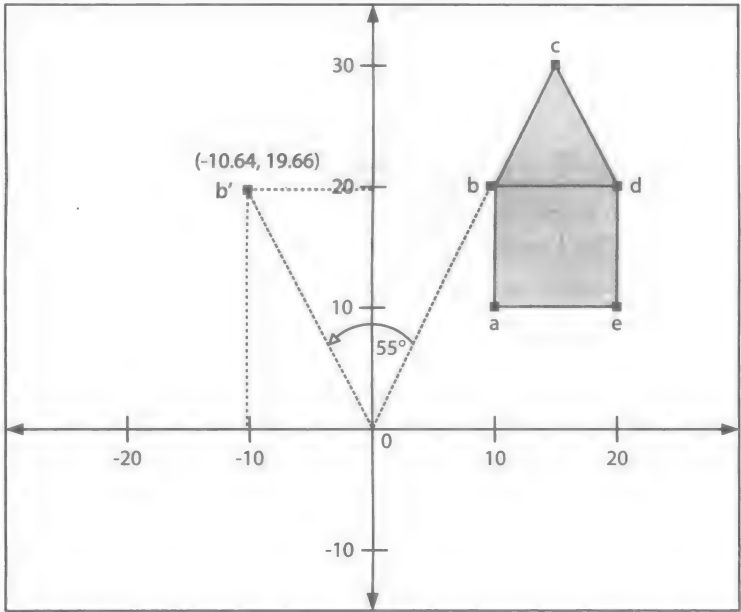


图 6-24 包括 5 个 $x-y$ 坐标点的二维虚拟对象。也表示了一个向量 b 旋转 55° 后的新向量 b'

为了让“房子”旋转 55 度 (55°)，我们必须和 b 一样将图中每个向量都旋转 55° 。旋转是一个坐标 (X, Y) 经过一个旋转角 β 到一个新坐标 (X', Y') 的线性变化。对于一个正的 β ，旋转是顺时针方向，如果 β 是负的，它就是逆时针方向。公式 (6-14) 表示了二维旋转的表达式。

$$\begin{aligned} X' &= \cos\beta * X - \sin\beta * Y \\ Y' &= \sin\beta * X - \cos\beta * Y \end{aligned} \tag{6-14}$$

在这个图中，从坐标原点 $(0, 0)$ 到 $(X, Y) = (10, 10)$ 的向量 b 转换到从坐标原点 $(0, 0)$ 到 $(X', Y') = (-10.64, 19.66)$ 的向量 b' 。公式 (6-15) 给出了这种转换的计算。

$$\begin{aligned} X' &= \cos 55^\circ * 10 - \sin 55^\circ * 20 \\ &= -10.64 \\ Y' &= \sin 55^\circ * 10 - \cos 55^\circ * 20 \\ &= 19.66 \end{aligned} \tag{6-15}$$

254

1. CORDIC 算法

CORDIC 算法可以用来执行三角、双曲线、对数、指数、平方根等功能。CORDIC 算法可用于袖珍计算器和 2D/3D 图形处理器的设计。此外，已经证实可以开发出简单的 CORDIC 算法，这是一种迭代算法，仅需要简单的功能，比如整数加法、减法和算术右移 [7]。右移用来表示整数除以 2、4、8 等。下一步将介绍迭代 CORDIC 算法可以用来计算线性变换 (公式 (6-14))。也表示简单图形处理器的数据通路的设计。

公式 (6-14) 中的表达式需要复杂的 cosine 和 sine 函数。然而，如公式 (6-16) 所示，它们可以通过分解两个表达式中的 $\cos \beta$ 得到简化。你将会知道，线性变换所需的计算 $\tan \beta$ 和保持 $\cos \beta$ 从计算中分离会更容易一些。然而， $\cos \beta$ 的结果将被用作一个比例因子来调整每个新产生的坐标点。

$$\begin{aligned} X' &= \cos\beta * (X - \tan\beta * Y) \\ Y' &= \cos\beta * (\tan\beta * X + Y) \end{aligned}$$

(6-16)

这种简单的迭代算法只使用一组固定旋转角的正切值。表 6-12 列出了 7 个角度，45°、27°、14°、7° 等，正切值分别等于 1、1/2、1/4、1/8 等。对于整数运算，表中的每个角度都被凑为最近的整数。

表 6-12 7 个 $\tan \beta$ 值和与它们相应的 β 近似值

| i | $2^{-i} = \tan \beta_i$ | $\beta_i = \tan^{-1} 2^{-i}$ |
|-----|-------------------------|------------------------------|
| 0 | 1 | 45° |
| 1 | 1/2 | 27° |
| 2 | 1/4 | 14° |
| 3 | 1/8 | 7° |
| 4 | 1/16 | 4° |
| 5 | 1/32 | 2° |
| 6 | 1/64 | 1° |

一个向量，如图 6-24 中的 b ，可以通过 4 步完成旋转角度 $\beta = 55^\circ$ ，第 1 步旋转向量 45°，然后是 7°，然后是 2°，最后是 1°，如公式 (6-17) 所示。第 4 步中的结果包含比例因子 $0.701 = \cos 1^\circ * \cos 2^\circ * \cos 7^\circ * \cos 45^\circ$ 。除去比例因子，值 $X' = -15.1$ 和 $Y' = 28.1$ 在常数 1.427 (1/0.701) 下是最大的。不包含这个得到的 1.427 的 X' 和 Y' 最终值在第 5 步给出。这些值与公式 (6-15) 中得到的那些值间的一点不同是由人工计算伴随的舍入误差引起的。

255

第 1 步:

$$\begin{aligned} X' &= \cos 45^\circ * (10 - 1 * 20) // (\tan 45^\circ = 1) \\ &= \cos 45^\circ * (-10) \\ Y' &= \cos 45^\circ * (1 * 10 + 20) \\ &= \cos 45^\circ * (30) \end{aligned}$$

第 2 步:

$$\begin{aligned} X' &= \cos 7^\circ * \cos 45^\circ * (-10 - 1/8 * 30) // (\tan 7^\circ = 1/8) \\ &= \cos 7^\circ * \cos 45^\circ * (-13.75) \\ Y' &= \cos 7^\circ * \cos 45^\circ * (1/8 * -10 + 30) \\ &= \cos 7^\circ * \cos 45^\circ * (28.75) \end{aligned}$$

第 3 步:

$$\begin{aligned} X' &= \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (-13.75 - 1/32 * 28.75) // (\tan 2^\circ = 1/32) \\ &= \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (-14.65) \\ Y' &= \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (1/32 * -13.75 + 28.75) \\ &= \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (28.32) \end{aligned}$$

第 4 步:

$$\begin{aligned} X' &= \cos 1^\circ * \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (-14.65 - 1/64 * 28.32) // (\tan 1^\circ = 1/64) \\ &= \cos 1^\circ * \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (-15.1) \\ Y' &= \cos 1^\circ * \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (1/64 * -14.65 + 28.32) \\ &= \cos 1^\circ * \cos 2^\circ * \cos 7^\circ * \cos 45^\circ * (28.1) \end{aligned}$$

$$\begin{aligned}
 \text{第 5 步: } X' &= 0.701 * (-15.1) \\
 &= -10.59 \approx -10.54 (\text{Eq. (6.15)}) \\
 Y' &= 0.701 * (28.1) \\
 &= 19.7 \approx 19.66 (\text{Eq. (6.15)})
 \end{aligned}$$

(6-17)

尽管这个示例说明了迭代过程可以消除任意角的正切值计算的需要，如 55° ，对于一个简单图形的数据通路，存在一些实现的复杂性，如下：

- 如何选择表 6-12 中的下一个 β_i 值。
- 何时结束计算。
- 对于一个给定的旋转角应该使用什么比例因子。

一个解决所有这些实现复杂性问题的方案是使用固定数量的步骤，与目标旋转角度无关 [7]。如果之前步骤的结果出现了超出，对相反方向旋转的需要是有必要的。比如，向量旋转 55° 需要使用 45° 、 27° 、 -14° 、 -7° 、 4° 、 2° 和 -1° 七个步骤。而且，对于所有的目标旋转角度仅需要一个比例因子 $= 0.6048$ (公式 (6-18))。注意，因为 $\cos \beta_i = \cos -\beta_i$ ，这个仅有的比例因子不受旋转方向的影响。步骤越多，比例因子越趋近于步骤为无穷大时的最大值 0.607。

$$0.6048 = \cos 1^\circ * \cos 2^\circ * \cos 4^\circ * \cos 7^\circ * \cos 14^\circ * \cos 27^\circ * \cos 45^\circ \quad (6-18)$$

对于一个简单图形的数据通路，算法的每一步都需要整数运算。公式 (6-17) 中每一个结果， $1/8 X$ 、 $1/32 X$ 等都由算术右移得到。例如， $1/8 * 10$ 的值，如果转换为最近的整数，等于右移 3 次后的 $10 = (01010)_2$ ，如下介绍。算术右移用于处理正的和负的坐标值。

$$\begin{aligned}
 1/8 * 10 &= (01010)_{2s} / 8 \\
 &= (01010)_{2s} > > > 3 \quad (> > > \text{表示一个算术右移运算符}) \\
 &= 1
 \end{aligned}$$

或

$$\begin{aligned}
 1/8 * -10 &= (10110)_{2s} / 8 \\
 &= (10110)_{2s} > > > 3 \\
 &= (11110)_{2s} \\
 &= -2
 \end{aligned}$$

下面描述在 -90° 和 $+90^\circ$ 之间的 β 值的旋转迭代算法 (即 $\beta \leq |90^\circ|$)。最终的新坐标值由公式 (6-19) 中的两个表达式决定。

$\beta \leq |90^\circ|$ 的迭代旋转算法

$$\begin{aligned}
 X_0 &= X \\
 Y_0 &= Y \\
 \beta_0 &= \beta \\
 A_0 &= 1; \\
 d_i &= +1 \text{ if } \beta_i \geq 0, \text{ 或 } d_i = -1 \text{ if } \beta_i < 0 \\
 X_{i+1} &= X_i - d_i (Y_i > > > i) \\
 Y_{i+1} &= d_i (X_i > > > i) + Y_i \\
 \beta_{i+1} &= \beta_i - d_i * \tan^{-1} 2^{-i} \quad //(表6-12) \\
 A_{i+1} &= A_i * \cos(\tan^{-1} 2^{-i}) \\
 \text{for } i &= 0, 1, 2, \dots, k-1
 \end{aligned}$$

256

257

$$\begin{aligned} X' &= A_k * X_k \\ Y' &= A_k * Y_k \end{aligned} \quad (6-19)$$

A_k 的值是一个比例因子, 是一个 FP 数字。例如, $k = 7$, $A_7 = 0.6048$ 。因此, 对于每一个向量, 一旦它最后得到的坐标点 (X_k, Y_k) 确定了, X_k 和 Y_k 都分别与常数 A_k 相乘, 去产生最新的坐标点 (X', Y') 。这就需要有一个 FP 乘数, 因此它将由 CPU 产生。公式 (6-20) 表示了 (X_{i+1}, Y_{i+1}) 当 $i = 0, 1$ 和 2 时的计算, 其中 $(X_0, Y_0) = (10, 20)$, $\beta_0 = 55^\circ$ 。表 6-13 列出了所有的当 $i = 0, 1$ 和 6 时的 X_{i+1} 、 Y_{i+1} 和 A_{i+1} 的值。

$$\begin{aligned} X_1 &= \cos 45^\circ * (10 - (1)(20 > > > 0)), d = 1 \quad // 55^\circ - 45^\circ = 10^\circ \\ &= 0.7071 * (10 - 20) \\ &= 0.7071 * (-10) \\ Y_1 &= \cos 45^\circ * ((1)(10 > > > 0) + 20) \\ &= 0.7071 * (10 + 20) \\ &= 0.7071 * (30) \\ X_2 &= \cos 27^\circ * \cos 45^\circ * (-10 - (1)(30 > > > 1)), d = 1 \quad // 10^\circ - 27^\circ = -17^\circ \\ &= 0.8910 * 0.7071 * (-10 - 15) \\ &= 0.6300 * (-25) \\ Y_2 &= \cos 27^\circ * \cos 45^\circ * ((1)(10 > > > 1) + 30) \\ &= 0.6300 * (-5 + 30) \\ &= 0.6300 * \cos 45^\circ * (25) \\ X_3 &= \cos(-14^\circ) * \cos 27^\circ * \cos 45^\circ * (25 - (-1))(25 > > > 2), d = -1 \\ &\quad // -17^\circ - (-14^\circ) = -13^\circ \\ &= 0.6113 * (-25 + 6) \\ &= 0.6113 * (-19) \\ Y_3 &= \cos(-14^\circ) * \cos 27^\circ * \cos 45^\circ * ((-1)(-25 > > > 2) + 25) \\ &= 0.6113 * (7 + 25) \\ &= 0.6113 * (32) \end{aligned} \quad (6-20)$$

258

表 6-13 描述图 6-24 中旋转 55° 坐标为 $(10, 20)$ 的向量 b 所得的中间结果

| | β_{i+1} | X_{i+1} | Y_{i+1} | d_i | A_{i+1} | $X_i - d_i(Y_i > > > i)$ | $d_i(X_i > > > i) + Y_i$ |
|---|-----------------------------------|------------------|----------------|-------|-----------|--------------------------|--------------------------|
| 0 | $10^\circ = 55^\circ - 45^\circ$ | $-10 = 10 - 20$ | $30 = 10 + 20$ | 1 | 0.7071 | $10 - (1)(20 > > > 0)$ | $(1)(10 > > > 0) + 20$ |
| 1 | $-17^\circ = 10^\circ - 27^\circ$ | $-25 = -10 - 15$ | $25 = 5 + 30$ | 1 | 0.6300 | $-10 - (1)(30 > > > 1)$ | $(1)(-10 > > > 1) + 30$ |
| 2 | $-3^\circ = -17^\circ + 14^\circ$ | $-19 = -25 + 6$ | $32 = 7 + 25$ | -1 | 0.6113 | $-25 - (-1)(25 > > > 2)$ | $(-1)(-25 > > > 2) + 25$ |
| 3 | $4^\circ = -3^\circ + 7^\circ$ | $-15 = -19 + 4$ | $35 = 3 + 32$ | -1 | 0.6067 | $-19 - (-1)(32 > > > 3)$ | $(-1)(-19 > > > 3) + 32$ |
| 4 | $0^\circ = 4^\circ - 4^\circ$ | $-17 = -15 - 2$ | $34 = -1 + 35$ | 1 | 0.6053 | $-15 - (1)(35 > > > 4)$ | $(1)(-15 > > > 4) + 35$ |
| 5 | $-2^\circ = 0^\circ - 2^\circ$ | $-18 = -17 - 1$ | $33 = -1 + 34$ | 1 | 0.6049 | $-17 - (1)(34 > > > 5)$ | $(1)(-17 > > > 5) + 34$ |
| 6 | $-1^\circ = -2^\circ + 1^\circ$ | $-18 = -18 + 0$ | $34 = 1 + 33$ | -1 | 0.6048 | $-18 - (-1)(33 > > > 6)$ | $(-1)(-18 > > > 6) + 33$ |

在第 7 步最后转换的坐标点是 $(X_7, Y_7) = (-18, 34)$, 包括得到的 $1/A_7 = 1.427$ 。最新的坐标点 (X', Y') 由 X_7 、 Y_7 分别乘以 $A_7 = 0.6048$ 得到, 如下所示:

$$\begin{aligned}
 X' &= 0.6048 * (-18) \\
 &= -10.88 \\
 Y' &= 0.6048 * (34) \\
 &= 20.56
 \end{aligned}
 \tag{6-21}$$

公式 (6-21) 中 $X' = -10.88$ 和 $Y' = 20.56$ 的值和公式 (6-15) 中得到的 -10.64 和 19.6 稍有不同, 这是由整数运算引起的。

对于旋转角度 $\beta > |90^\circ|$, 首先旋转 $\pm 90^\circ$ 或 $\pm 180^\circ$ 是有必要的。例如, $\beta = 125^\circ$, 先旋转 90° 会让目标旋转角度减少到 35° , 这样就 $< 90^\circ$ 。因为 $\cosine \pm 90^\circ = 0$, $\sin \pm 90^\circ = \pm 1$, 最初的 $\pm 90^\circ$ 旋转会把原始的 X_0 、 Y_0 和 β_0 值改变如下:

$$X_0 = -d * Y$$

$$Y_0 = d * X$$

$$\beta_0 = \beta - d * 90 \quad \text{其中, 如果 } \beta \geq 0 \text{ } d = 1, \text{ 或者, 如果 } \beta < 0 \text{ } d = -1$$

相对地, 最初 180° 的旋转将减少 $\beta = 125^\circ$ 的值为 $-55^\circ > -90^\circ$, 原始的 X_0 、 Y_0 和 β_0 值改变如下:

$$X_0 = -X$$

$$Y_0 = -Y$$

$$\beta_0 = \beta - d * 180 \quad \text{其中, 如果 } \beta \geq 0 \text{ } d = 1, \text{ 或者, 如果 } \beta < 0 \text{ } d = -1$$

259

对于 $\beta > |180^\circ|$ 和 $\leq |360^\circ|$, 最初必要的 $\pm 360^\circ$ 的旋转会将 β 减少到 $< |180^\circ|$ 。然而, 这样会保持原始值和如下一样:

$$X_0 = X$$

$$Y_0 = Y$$

$$\beta_0 = \beta - d * 360 \quad \text{其中, 如果 } \beta \geq 0 \text{ } d = 1, \text{ 或者, 如果 } \beta < 0 \text{ } d = -1$$

最后, 对于 $\beta > |360^\circ|$, β 由 $\beta \bmod 360$ 替代。即

$$X_0 = X$$

$$Y_0 = Y$$

$$\beta_0 = \beta \bmod 360 \quad \text{如果 } \beta \geq |360^\circ|$$

如下所示的伪代码指定旋转一个给定角度为 β 的虚拟对象所需的步骤。

迭代旋转伪代码:

```

object_transform(x[], y[], β, n, x'[], y'[])
  if β > |360| then
    β = β mod 360;
  endif
  if β > |180| then
    if β > 0 then
      β = β - 360;
    else
      β = β + 360;
    endif
  endif
  d = 1;
  if β > |90| then
    d = -1;
    if β > 0 then
      β = β - 180;
    else
      β = β + 180;
    endif
  endif

```

```

endif
//Rotate n vectors
for i = 0 to n-1
    x0 = d * x[i];
    y0 = d * y[i];
    (x7, y7) = vector_transform(x0, y0,  $\beta$ ); //  $-90^\circ \leq \beta \leq 90^\circ$ 
    x'[i] = 0.6048 * x7; //apply the scaling factor A7
                                //= 0.6048
    y'[i] = 0.6048 * y7;
endfor
end

vector_transform(x0, x0,  $\beta_0$ ) //use the Iterative Rotation
Algorithm
    x = x0;
    y = y0;
     $\beta = \beta_0$ 
    If ( $\beta \geq 0$ )
        d = 1;
    else
        d = -1;
    for (i = 0; i < 7, i++)
        x = x - d * (y >>> i);
        y = d * (x >>> i) + y
         $\beta = \beta - d * \text{LUT}[i]$ ; //Look-up table <tn>Table 6.12,
                                //column 3
    endfor
    return(x, y);
end

```

260

2. 流水线数据通路和控制

典型地，一个虚拟对象包括数以千万计的坐标点，当这个对象要被旋转一定角度时所有的坐标点都要转换到一个新的坐标点。流水线数据通路实现上述“向量转换”功能中给出的伪代码可以在短时间甚至实时地处理许多向量。一个非流水线数据通路，比如一个多周期数据通路，与流水线数据通路比起来表现出较低的吞吐量，但会需要较少的硬件。

在一个非流水线数据通路中，这些增量角度必须存在一个查找表（LUT）中，它们将每次被读取一个用来计算下一步的旋转角度。另外，非流水线数据通路在迭代 i 过程中可能需要使用组合移位电路（第3章介绍过）去移位 X_i 和 Y_i 的值。

图 6-25 表示了一个实现“向量转换”功能的七阶段的流水线数据通路。注意， $-90^\circ \leq \beta \leq 90^\circ$ 。每一个流水线阶段表现出 7 个坐标转换步骤中的一个，并由表 6-13 中的例子表示了出来。每个阶段包含三个 2 的补码加法器 / 减法器模块。传入的 2 的补码旋转角度（Bin）的符号用来确定方向信号 d 的值，而 d 用于计算下个阶段所需的旋转角度 B_{out} 和新的坐标值 X_{out} 和 Y_{out} 。

每个阶段也需要输入 X_{in} 和 Y_{in} 算术右移后的值 SX_{in} 和 SY_{in} 。注意在流水线数据通路中，没有用于产生值 SX_{in} 和 SY_{in} 的电路。它们是硬布线右移。第一个阶段负责将最初的坐标点 (X_{in}, Y_{in}) 转换 45° ；第二个阶段负责将它输入的坐标点转换 27° ；等等。

3. HDL 模型

例 6-6 中的 HDL 代码描述了图 6-25 中的流水线数据通路。表 6-12 中的第 3 列指定了这 7 个旋转角度。所有的阶段表现出一样的功能，因此仅需一个控制信号来使能所有的流水线寄存器。

例 6-6 这里表示了图 6-25 中 7 个阶段流水线数据通路的 HDL 行为描述。

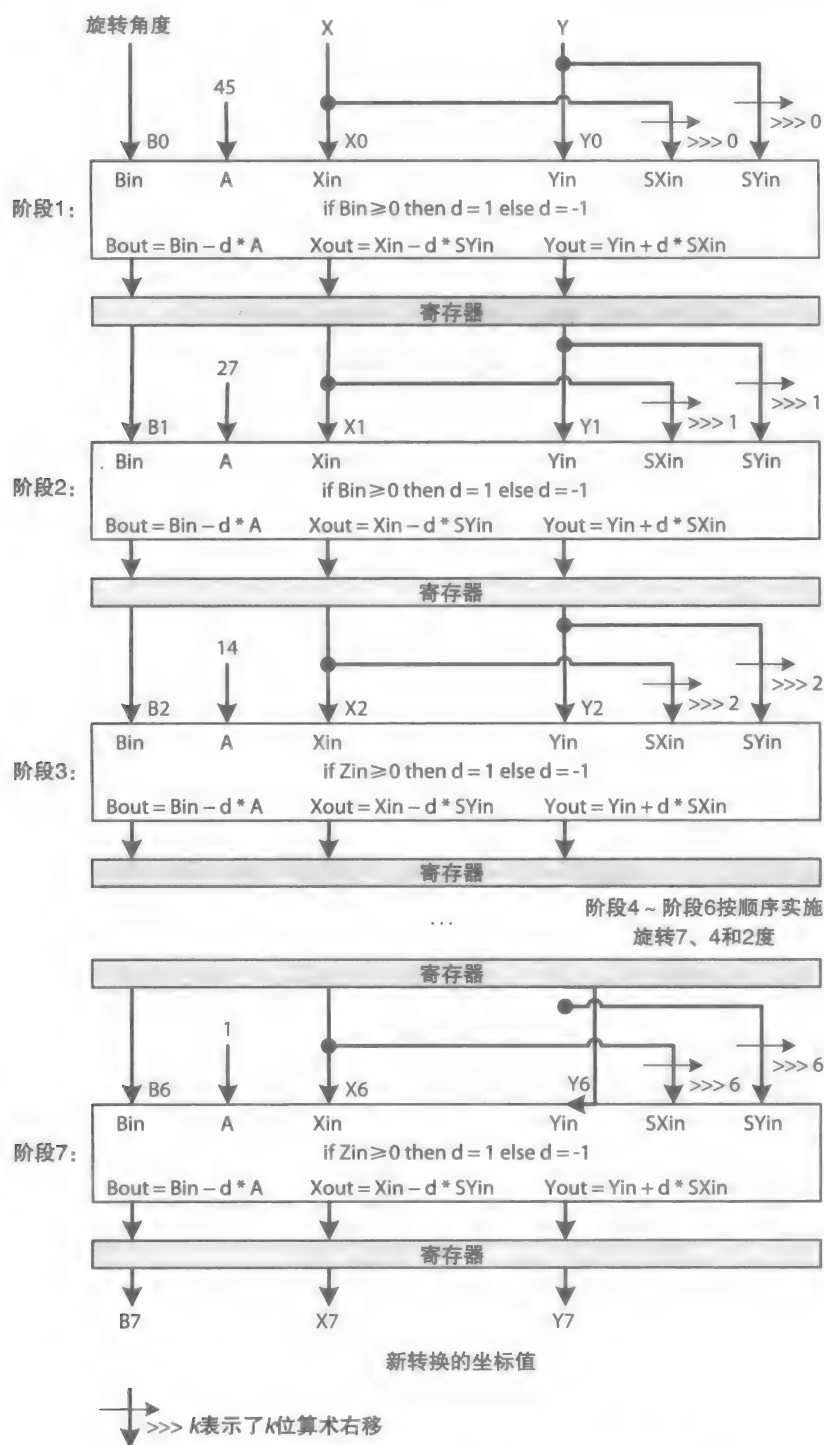


图 6-25 实现二维线性转换的七阶段流水线数据通路

解：cordic 模块是结构描述，而 stage 和 register 模块是行为描述（即使用 6.5.1 节所介绍的选择 II 来设计模块）。而且，为了简化 cordic 模块的描述，stage 模块和 register 模块是通过使用它们的端口名字被实例化，而不是它们的端口位置。

```

module cordic(
    input clock, reset, enable,
    input [7:0] xin, yin,
    input [7:0] degrees,
    output [7:0] cordxp, cordyp
);

wire [7:0] b0, x0, y0, b0o, x0o, y0o, sx0o, sy0o;
wire [7:0] b1, x1, y1, b1o, x1o, y1o, sx1o, sy1o;
wire [7:0] b2, x2, y2, b2o, x2o, y2o, sx2o, sy2o;
wire [7:0] b3, x3, y3, b3o, x3o, y3o, sx3o, sy3o;
wire [7:0] b4, x4, y4, b4o, x4o, y4o, sx4o, sy4o;
wire [7:0] b5, x5, y5, b5o, x5o, y5o, sx5o, sy5o;
wire [7:0] b6, x6, y6, b6o, x6o, y6o, sx6o, sy6o;
wire [7:0] b7, x7, y7, b7o, x7o, y7o, sx7o, sy7o;

assign b0 = degrees; //target rotation angle in degrees
assign x0 = xin; //initial coordinate value X
assign y0 = yin; //initial coordinate value Y
assign cordxp = x7o; //final computed coordinate value X7
assign cordyp = y7o; //final computed coordinate value Y7

//Stage 1: atan(2^0) = 45 degrees -----
stage s1(
    .b(b0), .x(x0), .y(y0), .atan(8'd45), .sx(x0), .sy(y0),
    .bp(b1), .xp(x1), .yp(y1));
pregregister1(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b1), .xin(x1), .yin(y1),
    .sxin({x1[7], x1[7:1]}), .syin({y1[7], y1[7:1]}),
    .bout(b1o), .xout(x1o), .yout(y1o), .sxout(sx1o), .syout(sy1o));

//Stage 2: atan(2^-1) = 27 degrees -----
stage s2(
    .b(b1o), .x(x1o), .y(y1o), .atan(8'd27), .sx(sx1o), .sy(sy1o),
    .bp(b2), .xp(x2), .yp(y2));

pregregister2(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b2), .xin(x2), .yin(y2),
    .sxin({2{x2[7]}, x2[7:2]}), .syin({2{y2[7]}, y2[7:2]}),
    .bout(b2o), .xout(x2o), .yout(y2o), .sxout(sx2o), .syout(sy2o));

//Stage 3: atan(2^-2) = 14 degrees -----
stage s3(
    .b(b2o), .x(x2o), .y(y2o), .atan(8'd14), .sx(sx2o), .sy(sy2o),
    .bp(b3), .xp(x3), .yp(y3));
pregregister3(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b3), .xin(x3), .yin(y3),
    .sxin({3{x3[7]}, x3[7:3]}), .syin({3{y3[7]}, y3[7:3]}),
    .bout(b3o), .xout(x3o), .yout(y3o), .sxout(sx3o), .syout(sy3o));

//Stage 4: atan(2^-3) = 7 degrees -----
stage s4(

```

```

        .b(b3o), .x(x3o), .y(y3o), .atan(8'd7), .sx(sx3o), .sy(sy3o),
        .bp(b4), .xp(x4), .yp(y4));
pregregister4(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b4), .xin(x4), .yin(y4),
    .sxin({4{x4[7]}}, x4[7:4]), .syin({4{y4[7]}}, y4[7:4]),
    .bout(b4o), .xout(x4o), .yout(y4o), .sxout(sx4o), .syout(sy4o));

//Stage 5: atan(2^-4) = 4 degrees -----
stage s5(
    .b(b4o), .x(x4o), .y(y4o), .atan(8'd4), .sx(sx4o), .sy(sy4o),
    .bp(b5), .xp(x5), .yp(y5));
pregregister5(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b5), .xin(x5), .yin(y5),
    .sxin({5{x5[7]}}, x5[7:5]),
    .syin({5{y5[7]}}, y5[7:5]),
    .bout(b5o), .xout(x5o), .yout(y5o), .sxout(sx5o), .syout(sy5o));

//Stage 6: atan(2^-5) = 2 degrees -----
stage s6(
    .b(b5o), .x(x5o), .y(y5o), .atan(8'd2), .sx(sx5o), .sy(sy5o),
    .bp(b6), .xp(x6), .yp(y6));
pregregister6(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b6), .xin(x6), .yin(y6),
    .sxin({6{x6[7]}}, x6[7:6]),
    .syin({6{y6[7]}}, y6[7:6]),
    .bout(b6o), .xout(x6o), .yout(y6o), .sxout(sx6o), .syout(sy6o));

//Stage 7: atan(2^-6) = 1 degrees -----
stage s7(
    .b(b6o), .x(x6o), .y(y6o), .atan(8'd1), .sx(sx6o), .sy(sy6o),
    .bp(b7), .xp(x7), .yp(y7));
pregregister7(
    .enable(enable),
    .clock(clock),
    .reset(reset),
    .bin(b7), .xin(x7), .yin(y7),
    .sxin({7{x7[7]}},
    .syin({7{y7[7]}},
    .bout(b7o), .xout(x7o), .yout(y7o), .sxout(sx7o), .syout(sy7o));

endmodule

module stage(
    input [7:0] b,
    input [7:0] x,
    input [7:0] y,
    input [7:0] atan,
    input [7:0] sx, sy,
    output reg [7:0] bp, xp, yp
);

```

//b is the 8-bit rotation angle between -90 and +90 degrees

```

//8-bit x-y coordinate point in a 2D space

wire mode = b[7]; //sign of rotation angle b

always@(*)
begin
    if (mode == 0)
        bp = b - atan; // b' = b - atan if b >= 0
    else
        bp = b + atan; //b' = b + atan if b < 0
end

always@(*)
begin
    if (mode == 0)
        xp = x - sy; //x' = x - y*(2-i) if b >= 0 for the
                        //ith iteration
    else
        xp = x + sy; //x' = x + y*(2-i) if b < 0
end

always@(*)
begin
    if (mode == 0)
        yp = y + sx; //y' = y + x * (2-i) if b >= 0
    else
        yp = y - sx; //y' = y - x * (2-i) if b < 0
end
endmodule

module preg(
    input enable,
    input clock, reset,
    input [7:0] bin, xin, yin,
    input [7:0] sxin, syin,
    output reg[7:0] bout, xout, yout, sxout, syout
);

always@(posedge clock or posedge reset)
begin
    if(reset == 1)
        begin
            bout <= 0;
            xout <= 0;
            yout <= 0;
            sxout <= 0;
            syout <= 0;
        end
    else if(enable == 1)
        begin
            bout <= bin;
            xout <= xin;
            yout <= yin;
            sxout <= sxin;
            syout <= syin;
        end
    end
end
endmodule

```

4. 仿真

例 6-6 中流水线的 Verilog 模型通过使用 Altera Quartus II、ModelSim 设计和仿真工具

进行了综合和仿真。例 6-7 描述了将图 6-24 中虚拟对象“房子”的 5 个向量转换 55° 的测试平台。仿真波形在图 6-26 中表示了出来。为了方便，仿真数据用十进制进行了表示。

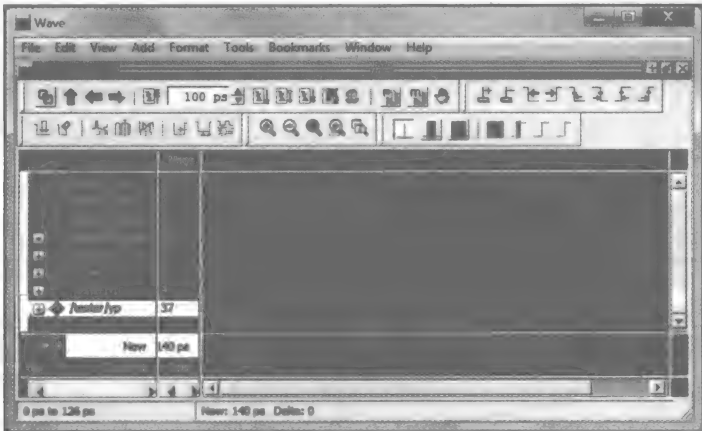


图 6-26 仿真例 6-6 中流水线描述的波形图；值都用十进制表示

例 6-7 这里表示了例 6-6 中仿真流水线模型的测试平台。

解：对象“房子”中只有 5 个向量，它们在代码中被列为 5 个测试向量。如果有很多测试向量时，我们可以从一个文件中读取这些测试向量。

```
module tester();
    reg clock, reset, enable;
    reg [7:0] degrees, x, y;
    wire [7:0] xp, yp;
    cordic    u1(clock, reset, enable, x, y, degrees, xp, yp);

    initial begin
        clock = 1;
        reset = 1;
        enable = 0;
        #5 reset = 0;
    end

    always begin
        #5 clock = ~clock;
    end

    initial begin
        enable = 1;
        degrees = 55; //rotate by 55 degrees
        x = 10; y = 10; //point a
        #10 x = 10; y = 20; //point b
        #10 x = 15; y = 30; //point c
        #10 x = 20; y = 20; //point d
        #10 x = 20; y = 10; //point d
        #80
        enable = 0;
        #20 $finish;
    end
endmodule
```

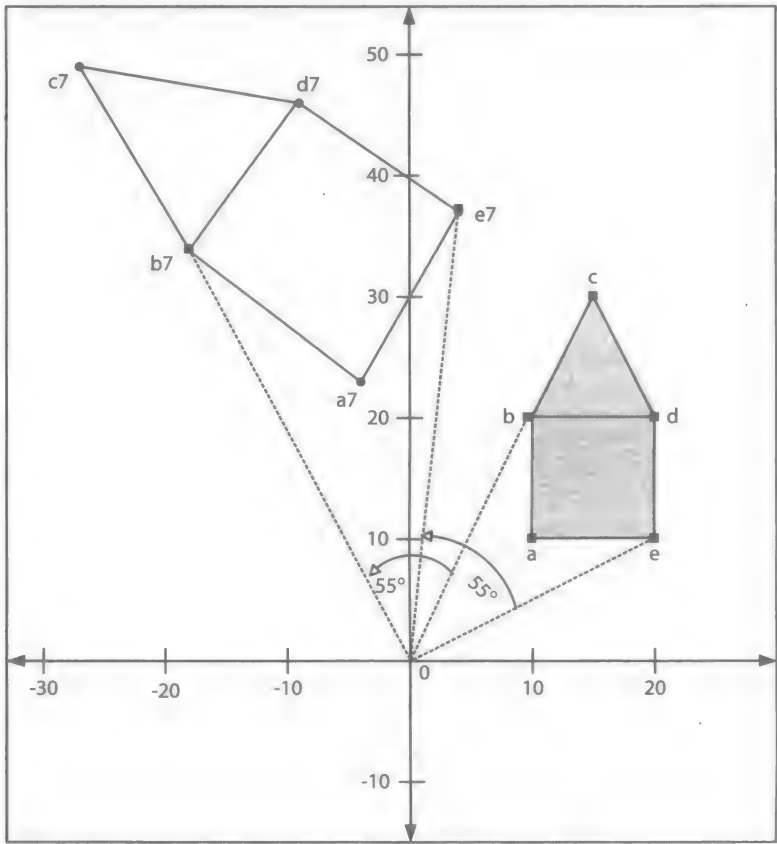
表 6-14 表示了从图 6-26 仿真波形图中获取到的原始和计算后的坐标点。新坐标值中包含一个等于 $1.653 = 1/0.6048$ 的增量，因此，旋转后的虚拟对象看起来比较大，如图 6-27 所示。为了去掉这个增量，每一个坐标值都必须乘以一个常数 0.6048。这需要一个 FPU，因

此这个尺度变换必须由 CPU 完成。

表 6-14 从图 6-26 仿真波形图中获取的仿真数据总结

| “房子” 原始坐标点 (十进制) | “房子” 转换后的坐标点 (十进制) |
|------------------|--------------------|
| a: (10, 10) | a7: (-4, 23) |
| b: (10, 20) | b7: (-18, 34) |
| c: (15, 30) | c7: (-27, 49) |
| d: (20, 20) | d7: (-8, 46) |
| e: (20, 10) | e7: (4, 37) |

* 坐标值包含一个等于 $1.653 = 1/0.6048$ 的增量



268

图 6-27 原始对象“房子”和它的 55° 旋转。新的对象被扩大 $1.653 = 1/0.6048$ 倍表示

表 6-15 中给出了不带增量和任何由公式 (6-14) 中转换表达式计算所得值的新得出的坐标值。这些得到的值与计算所得的值比较相近，但不一样。这种差异的原因是，这里给出的算法使用整数除法，从而比使用 FP 除法带来更多的舍入误差。

表 6-15 使用迭代旋转算法得到的新坐标值，而非使用公式 (6-14) 中转换表达式计算所得

| 向 量 | 计算得到的新坐标, 不含增量 = 1.653 | 计算得到的新坐标 |
|-----|------------------------|-----------------|
| a | (-2.42, 13.91) | (-2.46, 13.93) |
| b | (-10.88, 20.56) | (-10.64, 19.66) |

(续)

| 向 量 | 计算得到的新坐标, 不含增量 = 1.653 | 计算得到的新坐标 |
|-----|------------------------|-----------------|
| c | (-16.33, 29.64) | (-15.97, 29.49) |
| d | (-4.84, 27.82) | (-4.91, 27.85) |
| e | (2.42, 22.38) | (3.28, 22.11) |

CORDIC 旋转流水线可以通过带有两个内部存储单元的简单二维图形处理器来实现：1) 存储作为输入的虚拟对象的最初坐标点，2) 存储作为输出的计算得到的新坐标点。CORDIC 处理器将会是一个类似于图形处理器单元 (GPU) 的协处理器。然而，在这种情况下，这个协处理器将会提前使用它的内存空间执行固定的 CORDIC 旋转任务来绘图。协处理器将没有指令执行。CPU 将通过初始化虚拟对象的最初和最终的坐标点来启动这个协处理器，这些虚拟对象位于主存 (系统内存) 和每一个输入输出内部存储单元之间。一旦协处理器完成计算出新的坐标而且新的坐标被传送到主存中，协处理器将通知 CPU，这样将从主存中得到新的坐标，在每一个新坐标点乘以常数比例因子 0.6048 后，将在屏幕上显示出旋转后的虚拟对象。对于如何计算 CORDIC 处理器的吞吐量要参考练习 6.15。存储器的设计在下一章节中介绍，CPU 发起内存数据的大量传输在第 9 章中进行讨论。

参考文献

1. Intel Architecture Instruction Set Extensions Programming Reference, www.intel.com.
2. Steven Leibson and James Kim, Configurable processors: a new era in chip design, *IEEE Computer*, 2005, pp. 51-59.
3. SPEC CPU2006 and SPECviewperf from the Standard Performance Evaluation Corporation, <http://www.spec.org/>.
4. B. W. Bomar, Implementation of microprogrammed control in FPGAs, *IEEE Transactions on Industrial Electronics*, Vol. 49, No. 2, Apr 2002, 415-422.
5. Anantha Chandrakasan and Robert Broderson, Minimizing power consumption in digital CMOS circuits, *Proceedings of the IEEE*, Vol. 83, No. 4, 1995, 498-523.
6. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.
7. Ray Andraka, A survey of CORDIC algorithms for FPGA based computers, In: *Proc. ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays*, 191-200, 1998.

269

练习

- 对于练习 6.1 ~ 练习 6.3: 假设 8 位加法器的传播延时是 0.8ns，加法器 / 减法器是 1.1ns，2-1 选择器是 0.3ns，4-1 选择器是 0.6ns。而且，假设寄存器的建立时间 (τ_{st})、时钟到 q (τ_{cq}) 和时钟相位差 (τ_{cs}) 都是 0.05ns。
- 6.1 为以下每一个数据通路计算所需的最大时钟频率：
- a. 图 6-2 中单周期数据通路
 - b. 图 6-3 中多周期数据通路
 - c. 图 6-4 中流水线数据通路
- 6.2 计算通过练习 6.1 (a) 和练习 6.1 (b) 中数据通路能够计算出 $A + B + C \pm D$ 值所需的总时间。
- 6.3 估算在以下数据通路之间当生成 $N = 1000$ 次的 $A_i + B_i + C_i \pm D_i$ 值时的加速比，其中 $i = 0, 1, 2, \dots, 999$ 。忽略数据的读取和写入时延。

- a. 练习 6.1 (a) 与练习 6.1 (c)
- b. 练习 6.1 (b) 与练习 6.1 (c)
- 6.4 假设一个新处理器的电容性负载比旧处理器低 25%，工作时钟频率高 20%。确定两个处理器消耗的动态功率比值。对得到的结果做出评论。
- 6.5 假设新处理器的电压源是操作旧处理器所用的 50%，它的总体电容性负载低 15%，工作时钟频率高 40%。确定两个处理器消耗的动态功率比值。对得到的结果作出评论。
- 6.6 给出 3 位的 $A_value = 6$ 与 3 位的 $B_value = 5$ 相乘后的寄存器的内容，使用图 6-15 中多周期无符号乘法器。
- 6.7 设计满足如下建模的 8 位无符号乘法电路：
 - a. 使用原理图设计工具（比如 LogicWorks）或者全部结构式的 HDL 模型去建模这个乘法电路。你可以在数据通路中设计一个多功能寄存器去实现寄存器 A 、 B 和 P 。
 - b. 使用混合式 HDL 模型。对寄存器 A 、 B 、 P 和模 8 计数器使用行为建模。然后组合 A 、 B 、 P 、计数器和加法器来完成设计。
- 6.8 给出 4 位的 2 的补码 $A_value = 5$ 与 $B_value = -2$ 相乘后的寄存器的内容，使用图 6-21 中 2 的补码乘法器。
- 6.9 给出 4 位的 2 的补码 $A_value = -5$ 与 $B_value = -2$ 相乘后的寄存器的内容，使用图 6-21 中 2 的补码乘法器。

270

对练习 6.10 和练习 6.11：使用标准的反相不归零（NRZI）发生器 FSM（见第 5 章练习部分），设计出一个 NRZI 转换系统。假设输入流是一次处理的 16 位。而且，因为在源和目的模块之间没有共用的时钟，并且数据位在被称为 $D+$ 和 $D-$ 的双绞线上传输（其中 $D-$ 是 $D+$ 的反向），我们必须通过确保 NRZI 输出在多个时钟周期不会保持 1 或 0 的值来防止出现源和目的模块之间的数据同步问题。这是通过确保在输入流中每出现 6 个连续的 1 时，在输出端发生一次转变实现的。这样能保证输出流中最多会有 7 个连续的 1 或 7 个连续的 0。例如，对于得到的输入 $X = 1100011111110011$ (0xC7F3)，从右到左，改变后的 NRZI 发生器必须输出 $Y = 11101001111111011$ (或 0x1D3FB 从右到左)；对于 $X = 0xFFFF$ ， $Y = 0x3E03F$ ；对于 $X = 0xCFF6$ ， $Y = 0x123F8$ ；对于 $X = 0x0000$ ， $Y = 0xAAAA$ 。

NRZI 系统由一个数据通路和一个控制单元组成。完成以下练习：

- 6.10 设计一个数据通路包含 16 位的多功能（并行加载和右移）输入寄存器，一个标准 NRZI FSM，一个跟踪输入位的模 17 计数器。设计一个关于数据通路的基于 FSM 的控制单元，能够实现 NRZI，而且跟踪输入中 6 个连续的 1。（如果有必要，可能会用到一个 18 位的并行加载和右移寄存器来获取输出位。）
- 6.11 设计一个数据通路，由 16 位的多功能输入寄存器，一个标准 NRZI FSM，一个跟踪得到的输入位的模 17 计数器（CNTR1），一个跟踪输入中连续的 1 的模 7 计数器（CNTR2）组成。设计一个关于数据通路的基于 FSM 的控制单元，能够实现 NRZI，而且跟踪输入中 6 个连续的 1。（如果有必要，可能会用到一个 18 位的并行加载和右移寄存器来获取输出位。）
- 6.12 设计一个练习 6.11 中数据通路的微程序控制器。
- 6.13 计算图 6-24 中虚拟对象“房子”旋转 35° 后的新坐标点。将你的结果与用公式（6-14）中表达式计算出的结果进行对比。
- 6.14 计算图 6-24 中虚拟对象“房子”旋转 -35° 后的新坐标点。将你的结果与用公式（6-14）中表达式计算出的结果进行对比。
- 6.15 假设文中讨论过了实现七步流水线 CORDIC 旋转算法的二维图形处理器。而且，假设一个加法

器或减法器的时延是 0.8ns，寄存器的建立时间 (τ_{st})、时钟到 q (τ_{cq}) 和时钟相位差 (τ_{cs}) 都是 0.05ns。在 0.001 秒内流水线可以处理的坐标点的最大数量值大约是多少？忽略与读和写坐标点相关的时延。

- 6.16 用你自己选择的一种语言写一段程序来实现书中描述的 CORDIC 旋转伪代码。
- 6.17 以下定义了一个泰勒级数为 k 的指数函数：

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^{k-1}}{(k-1)!}$$

本系列中每一种情况都能从之前的情况中计算所得，这里介绍了前 4 种情况：

第一种情况 = 1

第二种情况 = 第一种情况 * $\frac{x}{1}$

271

第三种情况 = 第二种情况 * $\frac{x}{2}$

第四种情况 = 第三种情况 * $\frac{x}{3}$

使用一个组合电路加法器、乘法器、除法器和其他必要的模块，完成以下要求：

- a. 画一个多周期的数据通路来计算对给定 x 的 5 种情况的 e^x 。而且，确定在模块的时延分别为 τ_{st} 、 τ_{cq} 和 τ_{cs} 情况下的最小时钟周期。
- b. 画一个最小时延的流水线数据通路来计算对给定 x 的 5 种情况的 e^x 。而且，确定在模块的时延分别为 τ_{st} 、 τ_{cq} 和 τ_{cs} 情况下的最小时钟周期。

计算机安全

- 6.18 计算机安全（保密）：通过练习 11.17 来设计一个带有控制单元的流加密（也可以参考 11.5.1 节）。
- 6.19 计算机安全（保密）：通过练习 11.18 和 / 或练习 11.19 来理解 RSA 加密算法（11.5.3 节）。
- 6.20 计算机安全（保密）：通过练习 11.20 来理解非对称和对称密码（也可以参考 11.5 节）。
- 6.21 计算机安全（全面地理解加密哈希）：通过练习 11.21 ~ 练习 11.23（11.6 节和 11.7 节）。

272

存 储 器

7.1 简介

寄存器用于存储一个当我们需要时就可立即读出的数据。而存储器与之不同，存储器用于存储程序执行过程中的代码和数据。那些用于实现不同存储类型的存储技术所需要的硬件资源要比锁存器或触发器少得多。不过，存储器需要花费更多的时间来实现数据的存储（写）和检索（读）。

存储器的存储大小是以字节为单位的，1 字节即 8 位（b），表 7-1 展示了一系列存储单元。图 7-1 阐明了 1KB（1024B）存储单元的 2 种不同逻辑结构。在图 7-1a 中，1KB 存储单元中有 1024 个单元，每个单元中可存储 1B 内容，一个 1024×8 的存储单元需要 10 位地址（ $2^{10} = 1024$ ）来区分每一个 1B 单元。在图 7-1b 中，1024B 的存储单元含有 512 个位置，每个位置可存放 2B（16 位）数据，一个 512×16 的存储单元需要 9 位地址（ $2^9 = 512$ ）来区分每一个大小为 2B 的单元。

表 7-1 存储单元大小实例

| 单 元 | 读 法 | 实际大小 | 近似大小 |
|------|--------------|--|-------------|
| 1 KB | One kilobyte | $2^{10} = 1\,024$ bytes | 10^3 B |
| 1 MB | One megabyte | $2^{20} = 1\,048\,576$ bytes | 10^6 B |
| 1 GB | One gigabyte | $2^{30} = 1\,073\,741\,824$ bytes | 10^9 B |
| 1 TB | One terabyte | $2^{40} = 1\,099\,511\,627\,776$ bytes | 10^{12} B |

273

| 地 址 | | 数 据 |
|------|------------|----------|
| 十进制 | 二进制（10位） | 8位内容 |
| 0 | 0000000000 | 00010001 |
| 1 | 0000000001 | 10000111 |
| 2 | 0000000010 | 00111100 |
| 3 | 0000000011 | 11000000 |
| | ... | |
| | ... | |
| | ... | |
| | ... | |
| | ... | |
| | ... | |
| | ... | |
| 1023 | 1111111111 | 10000001 |

a) 1K×8存储器

| 地 址 | | 数 据 |
|-----|-----------|------------------|
| 十进制 | 二进制（9位） | 16位内容 |
| 0 | 000000000 | 0001000100010001 |
| 1 | 000000001 | 1000011111100001 |
| 2 | 000000010 | 0011110000001100 |
| 3 | 000000011 | 1100000011010100 |
| | ... | |
| | ... | |
| 511 | 111111111 | 1000000111001011 |

b) 512×16存储器

图 7-1 两种包含任意内容的 1KB 存储大小逻辑结构示意图 a) 1K×1B；b)512×2B

存储器数据读写的速度直接影响着冯·诺依曼结构计算机（如第 1 章图 1-2）的性能。近年来，计算机 CPU 运行速度的增长要比存储器读写数据速度的增长快得多，故而需要一

种更快存储技术、更优的存储组织结构来弥补计算机读写速度与 CPU 运算速度之间的差距。现在已经被使用的技术有管道技术和并行处理技术：管道技术通过并行存储读写操作和降低平均存储读写时间来增加系统的并发性，并行技术通过在更短的时间里传送更多数据来提升多处理机系统和实时应用的性能。

这一章大致介绍了已用的存储技术及其应用，全面讲述了存储单元内部结构、存储单元模型逻辑原理图以及在存储芯片内部用于支持多种应用的存储单元组织结构。这一章也阐述了包含当今已被广泛使用的存储技术在内的存储组织结构、存储时序以及存储器和其他部件交互的协议规范。同时，在本章中也探讨了关于现代计算机系统的存储结构，介绍了如何编写程序来降低存储使用次数和提升存储性能，并提供实例以供参考，此外，还提供了一个用硬件描述语言来控制存储的实例。

7.2 存储技术

通常，存储器可分为只读存储器（ROM）和随机存取存储器（RAM）。存储 1 位数据的存储器硬件被称为存储单元。在只读存储器中，存储单元中的数据是**非易失性**的，非易失性指的是存储单元中的数据内容在断电情况下也不会丢失。其他具有非易失性的存储器包括磁盘、flash 存储器 [1] 和光盘（CD-ROM），这三种非易失性存储器中数据都是以块为单位进行组织和存储读写的，磁盘存储器将在第 9 章详细介绍。

与只读存储器不同的是，随机存取存储器是**易失性**的，也就是说 RAM 存储单元中的数据内容在断电情况下会丢失。只读存储器和随机存取存储器都是随机访问的，也就是说两者访问一个存储单元中的数据所需要的时间是相同的，因为这个原因，只读存储器有时也被称作非易失性随机读写存储器（NVRAM）。

7.2.1 只读存储器

只读存储器的存储单元中的内容被固定为逻辑 0 或逻辑 1，可编程只读存储器（PROM）274使用熔线技术来一次性擦写存储器，存储单元可被擦写为逻辑 0（不熔断）或逻辑 1（熔断）。紫外线擦除可编程存储器（EPROM）现如今已不常用，它需要搁置在紫外线光源下一段时间（如 30 分钟）来擦除存储单元中的信息。另一种如今常用的电擦除可编程存储器（EEPROM）使用电子擦除来读写存储单元使之成为逻辑 0 或逻辑 1，然而电擦除可编程存储器只能被擦写编程一定次数（最少 100 000 次）。

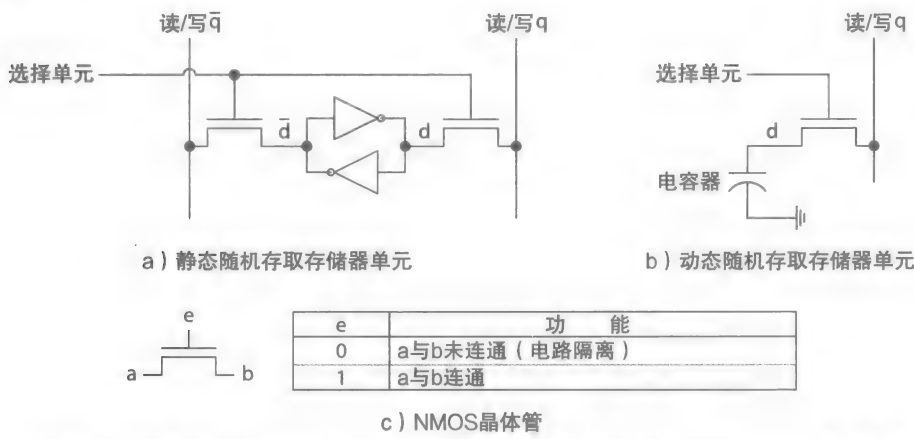
7.2.2 随机存取存储器

另一方面，随机存取存储器被设计为程序执行过程中存储代码和数据的主要功能部件。当一个逻辑 1 状态的随机存取存储器存储单元存储了一个静态电荷时，我们就称该存储器是**静态**的，只要存储器不断电，这个电荷将一直保留。静态随机存取存储器也被称为 SRAM。当一个逻辑 1 状态的随机存取存储器存储单元存储了一个动态电荷时，我们称该存储器为**动态**的，这个电荷在存储器不被刷新的情况下只会保留非常短的时间，典型的刷新频率是几毫秒一次，动态随机存取存储器也被称为 DRAM。

SRAM 与 DRAM 存储单元

图 7-2 说明了 SRAM 和 DRAM 的存储单元电路图。SRAM 的存储单元电路显示其需要两个晶体三级管和两个交叉耦合的非门，而 DRAM 只需要一个晶体三极管和一个小电容，

由此可见 DRAM 需要的硬件资源比 SRAM 要小得多。图 7-2c 中为 nMOS 的电路图，图中，晶体管的三个管脚被标记为 a 、 b 、 e 。 e 输入管脚为使能信号。当 e 管脚输入为 1 时，只要 a 、 b 管脚连接线路，无论从 a 到 b 还是从 b 到 a ，晶体管中的电子通路都是导通的，而当 e 管脚输入为 0 时，管脚 a 、 b 间的电路是隔离的（高阻抗）， a 、 b 管脚之间电流小到相当于两者电路未连接。



275 图 7-2 存储单元 a) SRAM 存储单元；b) DRAM 存储单元；c) nMOS 晶体管

在图 7-2a 中，当存储单元被选中时，两个晶体管将导通 d 到 q 和 \bar{d} 到 \bar{q} ，存储单元根据 q 和 \bar{q} 的信号将其内容读出或写入。但是当存储单元未被选中时，两个晶体管将保持交叉耦合的两个非门与 q 信号线和 \bar{q} 信号线为隔离状态，在这段时间中，只要存储单元不断电，存储单元将一直保持其中存储的数据 d 。

DRAM 的存储单元的读写操作与 SRAM 不同，在图 7-2b 中，当存储单元被选中并进行写操作时，电容器将被充电达到代表逻辑 1 的电压值，否则晶体管保持了电容器的孤立并使之不与 q 信号线直接相接。然而，当电容器充电至逻辑 1 的电压时，电容器中的电荷只能维持非常短的时间，一般为几毫秒。因此，电容器中的电荷必须周期性地在 一个刷新周期内被刷新。否则，电容器中的电荷将泄漏从而导致存储单元中的内容丢失，就像落在手电筒中的电池过段时间就会没电一样。

当存储单元被选中后，电容器将在 $\tau = R * C$ 秒内充电来达到代表逻辑 1 的电压。 R 指的是与晶体管被选中状态时相等价电阻的电阻值，单位是欧姆； C 指的是电容器的大小，单位是法。电容器中的电荷将在 $t = R_z * C$ 秒时间内释放，其中 R_z 是存储单元未被选中时的等价电阻的电阻值， R_z 要比 R 大得多。

[公式 $V_c = V_s (1 - e^{-t/RC})$ 说明了电容器是如何充电的，当充电电压源 V_s 为 5.0V（V 是伏特）时，电容器中的电压将在约 $t = 1RC$ 秒内达到 3.16V， $5.0 (1 - e^{-1})V = 3.16V$ 。当电压范围在 0 ~ 5.0V 时，3.16V 电压代表逻辑 1。然而电容器需要约 $t = 5RC$ 秒才能被充满，等式 $V = V_c \times e^{-t/R_z C}$ 说明了电容器是如何放电的 [2]。]

一个 DRAM 必须要在其存储单元丢失其信息内容之前进行刷新。以美国微光公司的 64MB 的 DRAM 为例 [3]，其组织结构是 $128M \times 4$ ，有 $512M (128M \times 4)$ 个存储单元，要求每个存储单元必须在 64ms 内被刷新。存储单元必须在刷新周期内被刷新从而留出足够的时间供 DRAM 在两个刷新操作之间进行读写操作。例如，微光公司的 DRAM 在一个更新周

期内每次同时刷新 64K (65 536) 个存储单元, 共需要 8192 (512M/64K) 次刷新操作来刷新所有的 512M 个存储单元, 每个刷新周期为 64ms。这意味着同时刷新 64K 个存储单元必须要在 7.8 μ s (64ms/8192) 内完成。而 DRAM 每次需要 7.5ns 来读写其内容, 因此, DRAM 在两次刷新操作之间能够进行多达 1000 (7.8 μ s/7.5ns) 次的读写操作。

在读操作中, 电容器中的电荷将决定该存储单元中存储的是 1 还是 0, 当电容器的电压在逻辑 1 的电压值时进行读写操作, 电容器中的电荷可能会丢失, 因此存储单元在进行读操作后必须被再次刷新。我们通过对刚刚进行读操作的存储单元再执行一次写操作来实现对该存储单元的再次刷新操作。

276

7.2.3 应用

有很多应用是针对只读存储器的, 例如, EEPROM 被用来存储系统上电时执行的引导装载程序, 或者存储基于 ROM 的可编程逻辑设备 (PLD) 的配置文件。EEPROM 技术也被用来设计 flash 存储器, 如闪存驱动器。然而, 因为存储在闪存中的数据是组织有序的, 并且以块的方式进行存取, 闪存的数据组织和读取方式与磁盘和光盘是相似的, 因此闪存的读取速度也相对较慢。当下许多便携式设备使用快闪存储器来代替磁盘驱动器。

SRAM 相对于 DRAM 需要更多的硬件资源, 因此其单位字节的成本要更高。但由于 SRAM 不需要 DRAM 的刷新周期和读后写周期, 其存取速度相对更快。现代计算机使用 DRAM 作为主存 (并不是全部采用 DRAM), 使用 SRAM 作为高速缓冲存储器 (cache), 从而减少了计算机读取数据消耗的平均时间。高速缓冲存储器的存储结构将在第 10 章详细描述。

7.3 存储单元阵列

所有的存储器内部都使用二维的存储单元组织结构, 这一结构有两大优点, 一是减少了存储器选择一系列目标存储单元所需要的信号线数量, 二是当存储器是动态随机存取存储器时, 存储器可同时刷新多个存储单元。在二维组织结构的存储器中每个存储单元需要两条信号线来确定, 而一维组织结构的存储器中每个存储单元只需要一条信号线, 但是二维组织结构的存储器需要的信号线数量要比一维组织结构少得多。例如, 一个 128B 的存储器中含有 1024 (128 \times 8) 个存储单元, 如果其组织结构是一维的, 如图 7-3a 所示, 选择不同的 1024 个存储单元需要 1024 条信号线, 每次选取一个存储单元。其存储单元被组织成 1K \times 1 \times 1 的单元阵列, 包含 1024 行、1 列, 存储单元在每个行和列的交叉处。

另一方面, 如图 7-3b 所示, 其存储单元被组织成 32 \times 32 \times 1 的单元阵列, 包含 32 行、32 列, 存储单元在每个行与列的交叉处。这一组织结构仅仅需要 64 (32 + 32) 条信号线来确定 1024 (32 \times 32) 个存储单元, 每次选择一个存储单元。同样, 一个 1M \times 1 的存储器如果其单元组织结构是一维的, 则其需要 1M (1 个非常大的数字) 条信号线, 而相对的, 若其组织结构是二维的, 只需要 2048 (2 \times 2¹⁰) 条信号线。很明显, 二维组织结构要远远优于一维组织结构。

在 32 \times 32 \times 1 单元阵列中, 只需要 32 个更新周期, 每次更新一行中全部 32 个存储单元; 相对的, 在 1K \times 1 \times 1 单元阵列中, 需要 1024 个更新周期, 每次只更新一个存储单元。当某一行选择信号线被选中时, 该行所有的存储单元都将同时被选中, 这一操作被称作行激活。此外, 由于行选择信号线是地址译码电路的输出, 其没有必要作为激活大量存储单元的直接输出, 相反, 信号线将使能一个晶体管来允许通过电源激活该行所有的存储单元。

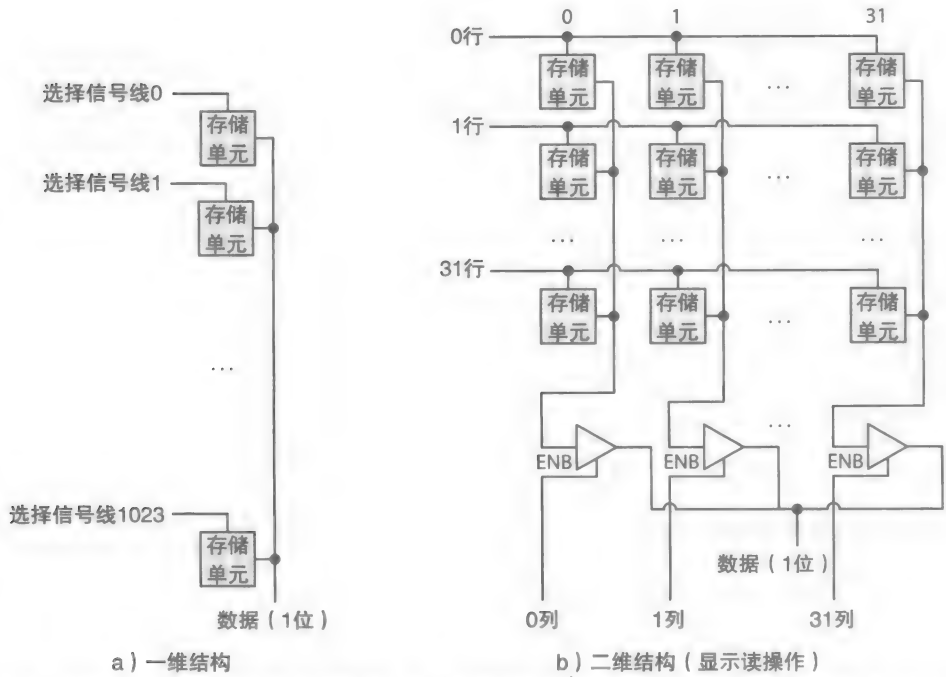


图 7-3 1024-存储单元内部组织结构: a) 一维组织结构需要 1024 条信号选择线; b) 二维组织结构需要 64 条信号选择线

存储器通过一个被称为感应放大器的专用电路将已选中行上的存储单元内容全部置为逻辑 1 或逻辑 0。感应放大器将存储单元的电压电平与参考电压相比较, 以此来判定是逻辑 1 或逻辑 0, 如参考电压的 50% 被用来表示逻辑 1。在一个读操作过程中, 如果存储单元中存储逻辑 0, 存储单元将轻微地拉低参考电压, 这将导致感应放大器察觉到逻辑 0。否则, 如果存储逻辑 1 的存储单元中的电压电平将轻微地拉高参考电压, 将导致感应放大器察觉到逻辑 1。

每一列只需要一个感应放大器。从感应放大器中输出出来的逻辑 0 或逻辑 1 将被锁存并提供给有列选择信号控制的三态缓冲器。当存储器为 DRAM 时, 锁存的输出信号也被用于刷新刚刚读过的存储单元。更多关于感应放大器的细节讨论已经超出了本书的范围, 可参考其他资料。

7.3.1 字存取

在图 7-4 中举例说明了一个 512×2 (一个 2 位字) 存储器的组织结构。在图中, 1024 个存储单元被分离为两个 $32 \times 16 \times 1$ 单元阵列, 实际上是一个 $32 \times 16 \times 2$ 的单元阵列。从这个单元阵列中每次存取两个存储单元信息, 分别从对两个 $32 \times 16 \times 1$ 单元阵列中各取一个。例如, 要求获取第 0 行第 0 列的选择信号将选中第 0 行第 0 列交叉处的存储单元, 同时也将选中第 0 行第 16 列交叉处的存储单元。正如图 7-3 所示, 在 $32 \times 16 \times 2$ 的单元阵列中, 单元阵列依旧是包含 32 行 32 列, 所以同样需要 32 个更新周期, 每次同时更新一行 32 个存储单元, 从而来更新所有的存储单元。

7.3.2 突发访问

突发访问指的是存储器传送突发数据的能力 (一次一个字)。突发数据的大小可能会比较小, 如几个字节, 也可能会比较大, 如一页 (通常为 4KB)。突发访问通过以下方式来存

取数据，先激活一行，然后按照一定的次序（如顺序的）依次选中列选择信号，一次一条，从而来读或者写对应的目标存储单元。每个突发访问必须先进行激活一行的操作。如果是存取一页或者以页模式存取，需要存取多行存储单元，存储器可能在当前行信息存取完毕时自动激活之后的行信号。页模式存取方式通过减少了存储器的空闲时间并同时使得整页信息无缝存取，从而提升了存储器的效率。

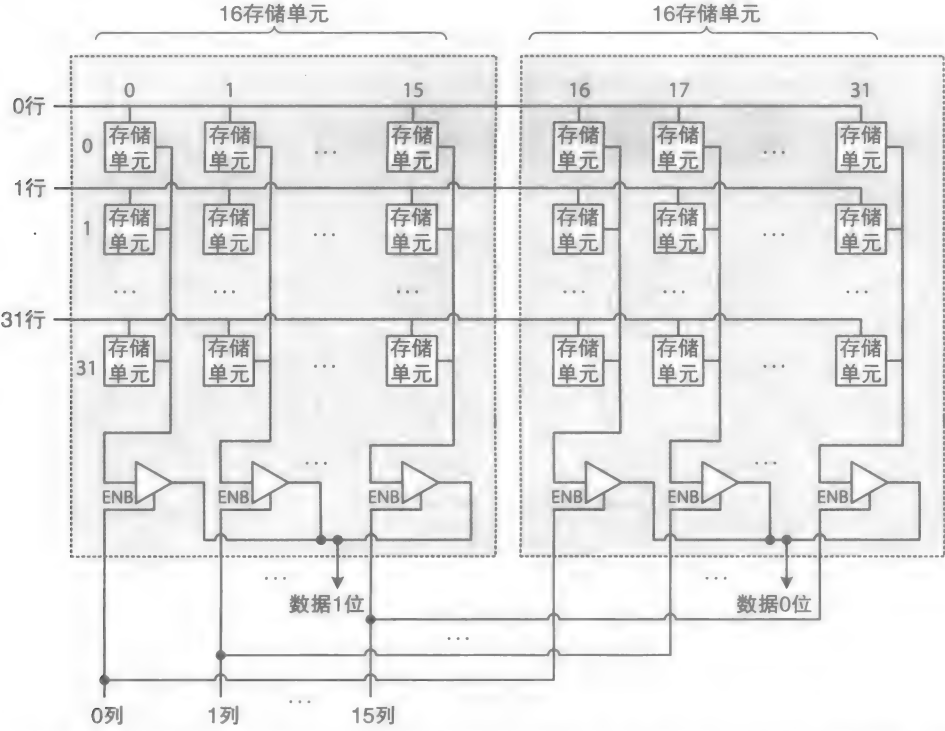


图 7-4 一个 1K 个存储单元组织成的 32×16×2 的单元阵列，生成一个 512×2 的存储器（显示读操作过程）

为了使存储器工作效率更高，存储单元可以被组织成存储片，每个存储片是一个单元阵列，如图 7-5 所示。在图中，一个 4K×1 的存储器由 4 个存储片构成，每个存储片是一个 32×32×1 的存储单元阵列。在这种情况下，在当前存储片上的存取操作仍在进行时，其他存储片上的行激活操作也可同时进行。因此，这使得当存储器存取数据操作涉及多个不同的存储片时，存储片存取的周转时间要少得多。

279

一个多存储片存储器可以设计支持在某存储片上正在进行块传输的同时，在另一存储片上进行短时间的间歇性的突发数据存取。在这种情况下，间歇性突发数据存取请求将中断正在某存储片上进行的块传输，允许在另一存储片上的短暂突发访问请求。

大多数现代存储芯片支持字存取和多存储片，同时芯片也支持突发访问和页模式存取。以微光公司的 512Mb（兆位）DRAM 存储器 [3] 为例，存储器可被设计为 128M×4 的 4 个存储片的 RAM，每个存储片为 8192×4096×4 的单元阵列，或者设计为 64M×8 的 4 个存储片的 RAM，每个存储片为 8192×2048×8 的单元阵列，也可被设计为 8M×16 的 4 个存储片的 RAM，每个存储片为 8192×1024×16 的单元阵列。例如微光的 128M×4 RAM，在单字为 4 位时，支持 1、2、4、8 字的突发访问，同时也支持页大小为 4096 位的页传输。

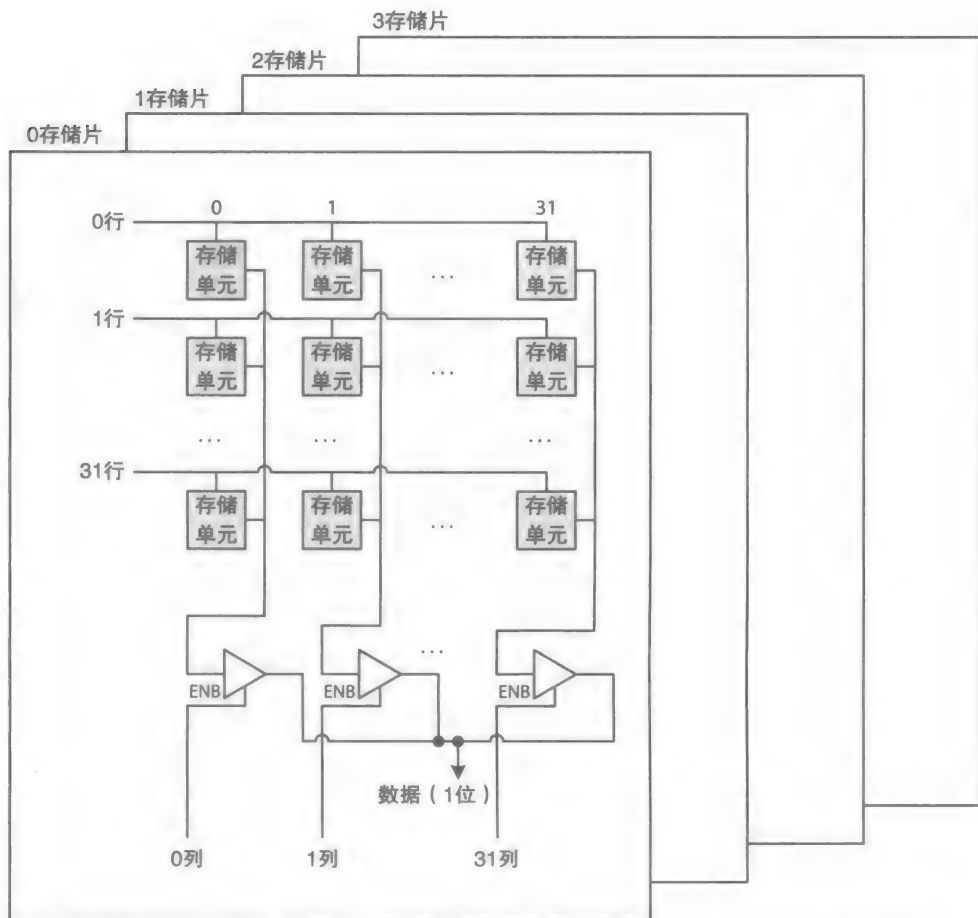


图 7-5 4K × 1 存储器组织结构，分成 4 个存储片，每个为 32 × 32 × 1 单元阵列

280

7.4 存储器组织结构

存储器的组织结构指的是在一个存储芯片内部组件和组件之间的组织关系，以及存储器之间交流通信的协议规范。使用多个存储器芯片来组成一个更大容量的存储器，称之为**存储器单元**。**地址总线、数据总线、控制总线**这三种总线信号是用来控制一个存储芯片或存储器单元的操作信号。其中，地址总线信号是用来说明如一个突发存取内容的起始地址或页的起始地址等一块单独的存储区的地址，地址总线被用来选中一系列目标存储单元以供读写操作。对于一个基本的存储器组织结构，控制总线是用来说明当前读写操作是读操作还是写操作，或者控制存储系统不在进行读写操作时使用数据总线来和其他部件进行交互。

图 7-6 阐述了 1K × 1 结构存储器的逻辑图以及其作为 SRAM 和 DRAM 时的方框图。SRAM 需要 10 位地址总线，标记为 a_0 到 a_9 ；1 位数据总线，标记为 d ；3 位低信号使能的控制总线，标记为 ce （芯片使能信号）、 we （写使能信号）、 oe （输出使能信号）。当芯片使能信号为低时，芯片被选中，可进行一系在该存储芯片的存储单元上的数据读写操作，其中，当 we 信号为 1 时（未选中），将执行读操作，而当 we 信号为 0 时（选中），将执行写操作。在读操作过程中，当输出使能信号 oe 为 0（选中）时，SRAM 中的数据将输出到数据总线上。

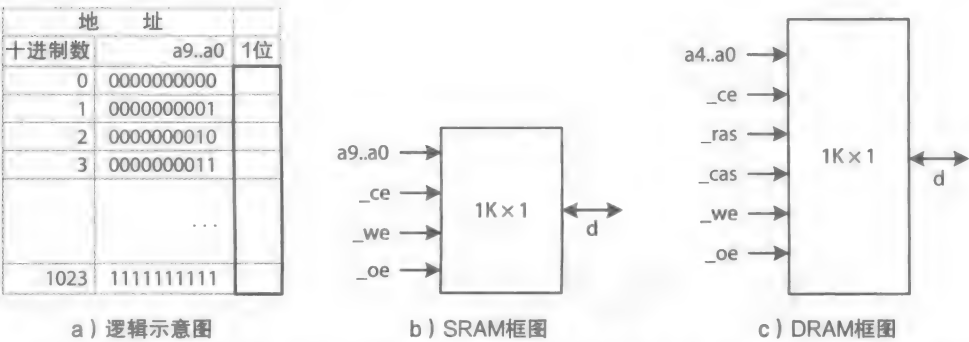


图 7-6 具有低使能控制信号的 1K × 1 存储器逻辑示意图和方框图：a) 逻辑示意图；b) SRAM 框图；c) DRAM 框图

相对于 SRAM 而言，DRAM 有更多的存储单元，并且，由于 DRAM 的存储单元需要刷新操作，其需要更多的控制信号线。其中，DRAM 的地址分为行地址和列地址，通过地址总线来传输，使用行地址选通（_ras）和列地址选通（_cas）来确定当前地址总线中的地址是行地址还是列地址。_ras 和 _cas 信号也被用来控制存储器进入刷新周期模式。在 1K × 1 的 DRAM 框图中，该存储器有 5 位地址总线，标记为 a_0 到 a_4 ，1 位数据总线被标记为 d 以及 5 位低信号使能的控制总线。

7.4.1 现代 DRAM

现代 DRAM 采用了同步运行设计，故而该芯片也被称为同步动态随机存取存储器（SDRAM）。芯片包含一个或多个管道化的数据通路，通过同时处理多个数据读 / 写请求，从而提高存储器的带宽。

281

典型的，一个现代 DRAM 芯片使用 _ce、_ras、_cas、_we 4 个接口信号形成一个 4 位的指令，这一指令也被称为存储器命令，使用存储器命令来选择并发送行地址、列地址以及存取模式（如单一访问或突发访问）等信息给存储器芯片，存储器命令也可被用来执行诸如启动刷新周期等存储器任务。图 7-7 展示了一个含有两片 SDRAM 的数据通路。存储器包含多个寄存器来装载存取模式以及行地址和列地址。两个存储片的行地址位宽（最高位或最低位）是相同的，而列地址是存储在一个计数器中，当存储器的存取模式表明该存取是一个突发访问或页传输时，存储在计数器中的列地址在每个时钟周期递增。

表 7-2 显示了一系列用于微光 SDRAM 的存储器命令。例如，如果命令为 $(0000)_2$ ，即 _ce=0、_ras=0、_cas=0、_we=0，SDRAM 将通过地址总线输入存取模式信息，存储模式信息之后将被载入模式寄存器，如图 7-7 所示。当存储器命令为 $(0011)_2$ 时，包含存储片号的行地址将根据存储片号装载入某个行地址寄存器。当存储器命令为 $(0101)_2$ 时，列地址将装入列地址计数器。行地址是用来激活目标存储片中的一行存储单元，而包含列地址的计数器是用来在存取模式为突发访问时自动生成列地址。

多年以来，人们对存储器的带宽以及存储系统种类的需求日益增加，导致产生了多种多样的存储器技术。在存储器芯片内部组织结构和交互协议中的进步促成了今天的高性能同步动态随机存取存储器（SDRAM）。例如，单速率 SDRAM 以一个时钟周期读写一次的速率存取信息，而双速率 SDRAM（DDR、DDR2、DDR3 等）以每个时钟读写两次的速率进行存取，双速率 SDRAM 在时钟的上升沿和下降沿分别进行存取操作。

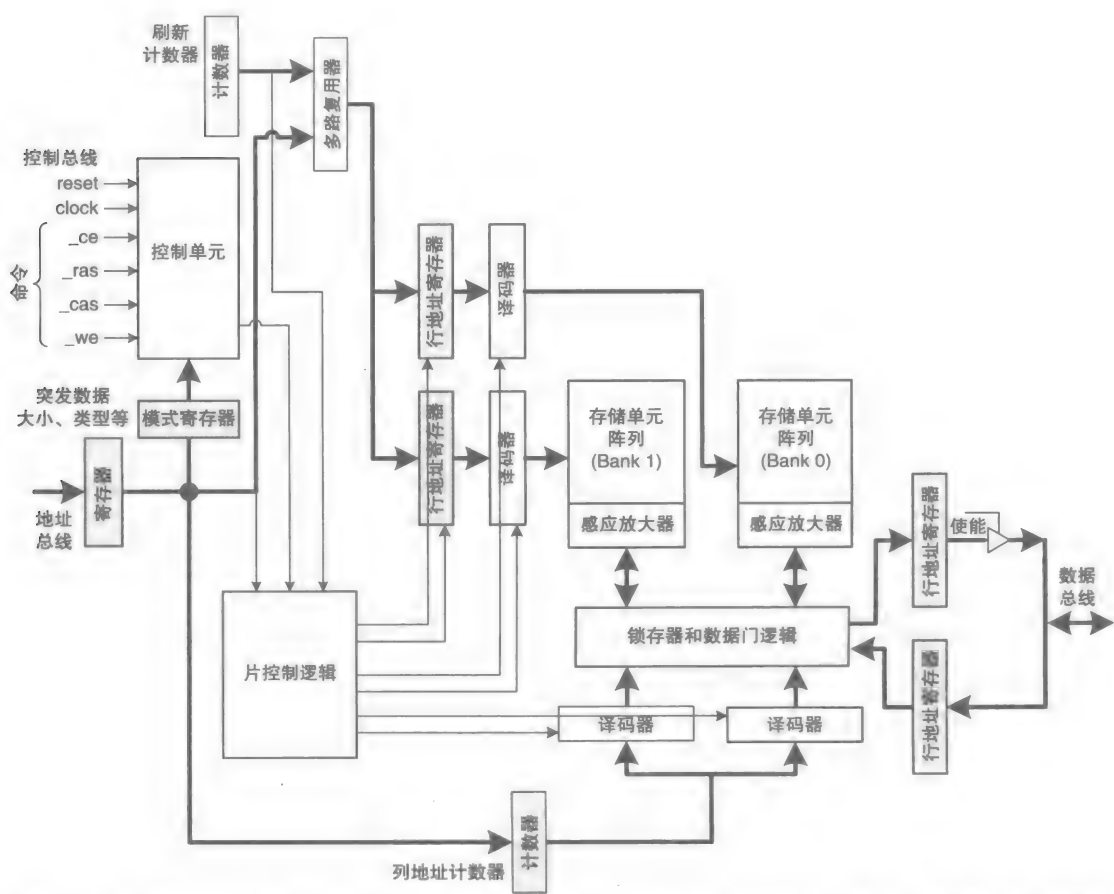


图 7-7 含有 2 个存储片的 SDRAM 内部组织结构，行地址信号选中了某一存储片，图中未显示所有的信号关系 [3]

表 7-2 微光 SDRAM 存储器命令样例

| 功 能 | 命 令 | | | |
|---------------------------|-----|------|------|-----|
| | _ce | _ras | _cas | _we |
| 芯片未选中 | 1 | d | d | d |
| 无操作 (NOP) | 0 | 1 | 1 | 1 |
| 输入下一行地址 (该行已激活) | 0 | 0 | 1 | 1 |
| 输入下一列地址并启动突发访问模式中的读操作 | 0 | 1 | 0 | 1 |
| 输入下一列地址并启动突发访问模式中的写操作 | 0 | 1 | 0 | 0 |
| 终止突发访问模式 (如有一个单独的读或写请求) | 0 | 1 | 1 | 0 |
| 使当前行无效 (为新一行做准备) | 0 | 0 | 1 | 0 |
| 启动刷新周期 | 0 | 0 | 0 | 1 |
| 通过地址信号线指定存取模式 (如突发访问或块访问) | 0 | 0 | 0 | 0 |

d 表示无需在意该位。

另一个例子是 Rambus 公司的专利技术，如 RDRAM 和 XDR DRAM 等，其采用了包传输的高速率点对点传输技术 [4]。在这个示例中，包指的是一个短的、突发的、1 比特位宽的、从一个源模块（如存储器）到目标模块（如处理器）的数据。包传输的通信模式和人类

使用信件交流的方式相似，每一封信（包）包含发件人地址（源地址）、收件人地址（目的地址）以及信件内容（数据），信件（包）在到达收件人地址（目的地址）前将经过一个或多个（点对点）邮局。

7.4.2 SRAM 存储单元模型

虽然一个真实的存储单元并不是通过逻辑门来构成，但是我们可以通过逻辑门来模拟 SRAM 存储单元的方式进而阐述存储器的设计和存储单元的操作。图 7-8a 阐述了一个 SRAM 存储单元结构原理的逻辑模型。这个存储单元由 SR 锁存器（无时钟）和三态缓冲门构成，两个电阻分别将两个三态缓冲门的输出端和地相连接，这个电阻也被称为下拉电阻，当三态缓冲门未使能时，这两个下拉电阻将使得存储单元的输出为 0 而不是高阻抗。这也会导致当存储单元未被选中时，锁存器的输入端 s 和 r 保持为 0，使得锁存器保持原有存储值 0 或 1 不变。该模型使用三态缓冲门来模拟图 7-2a 中的导通晶体管的功能，当存储单元未被选中时，SR 锁存器电路隔离，就像在真实的 SRAM 存储单元中使用交叉耦合的非门一样。图 7-8b 展示了存储单元的框图，图 7-8c 展示了一个下拉的三态缓冲门及其真值表。

282
283

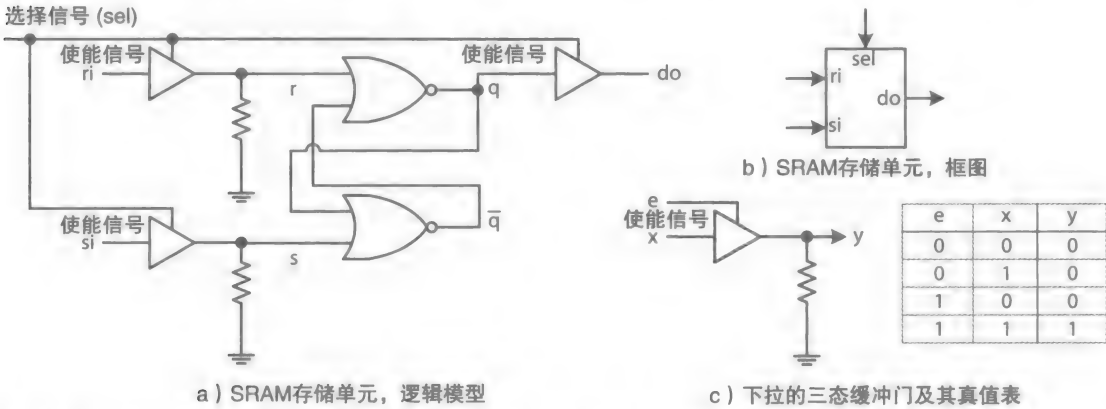


图 7-8 SRAM 存储单元逻辑模型：a) 门级模型；b) 存储单元框图；c) 下拉的三态缓冲门及其真值表

7.4.3 SRAM 芯片内部组织结构

虽然存储单元阵列是存储硬件的核心部分，但是存储单元阵列也需要一些额外的电路辅

以支撑，如转换给定的存储单元地址到具体的行地址和列地址的译码电路以及供数据进出存储单元阵列的数据通路，典型的是双向数据总线，双向数据总线减少了传输进出存储单元阵列的数据所需要的信号线数量。

例 7-1 设计一个 16×1 的 SRAM，要求 4 位的地址总线、1 位双向数据总线和 3 位控制总线。

解：如图 7-9 所示，在 SRAM 的逻辑图和方框图中，地址总线信号标记为 a_0 到 a_3 ，1 位数据总线信号标记为 d ，三

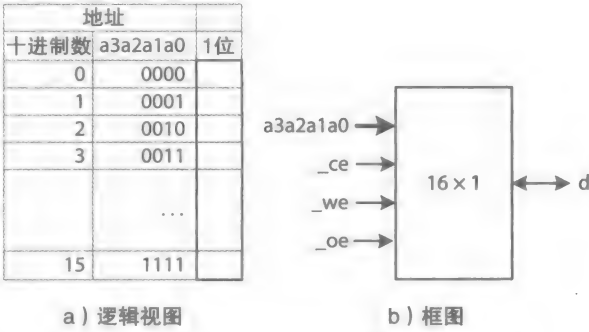


图 7-9 16×1 SRAM 逻辑视图和框图

位低使能控制总线信号标记为 _ce 、 _we 和 _oe 。设计的细节和读写操作过程将在之后描述。

图 7-10 展示了一个包含 $4 \times 4 \times 1$ 存储单元阵列和两个 2-4 译码器的 SRAM 内部组织结构。其中，行译码器转换地址信号的高两位 a_3 和 a_2 为 4 个行选择信号，列译码器转换地址信号的低位 a_1 和 a_0 为 4 个列选择信号。行译码器会一直保持使能状态以方便早期的行激活操作，但是列译码器必须在 $\text{_ce} = 0$ 的情况下才能被使能，正如图中所示。所有的信号线必须使用缓冲以避免信号源违规信号的输出。

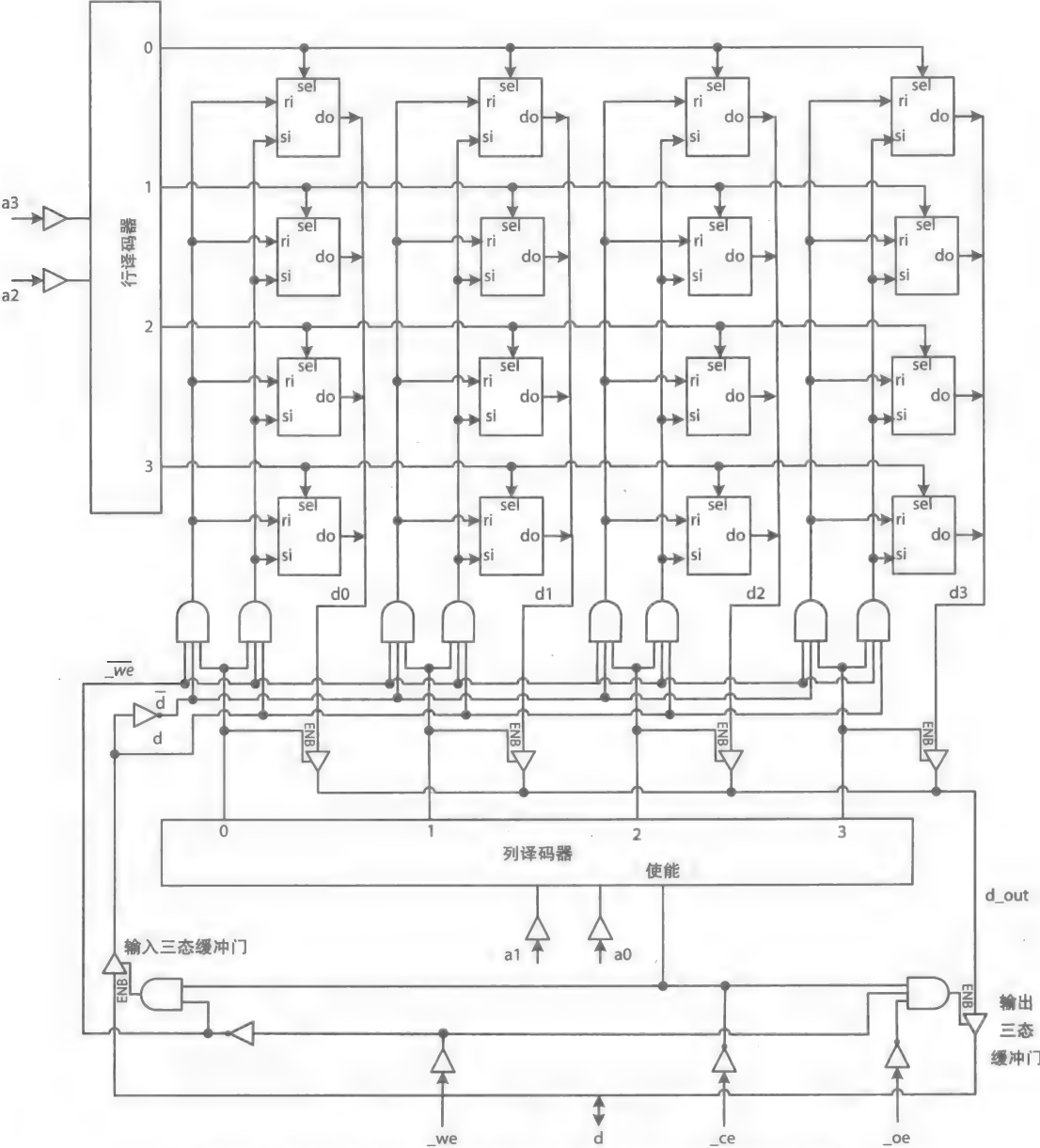


图 7-10 使用 $4 \times 4 \times 1$ 单元阵列的 16×1 SRAM 内部组织结构

在一个存储操作过程中，当 _ce 为 0 时，行选择激活信号将激活给定存储单元所在行上的所有相关的 4 个存储单元，通过使能该行存储单元的所有输出三态缓冲门，使得该行存储

284
285

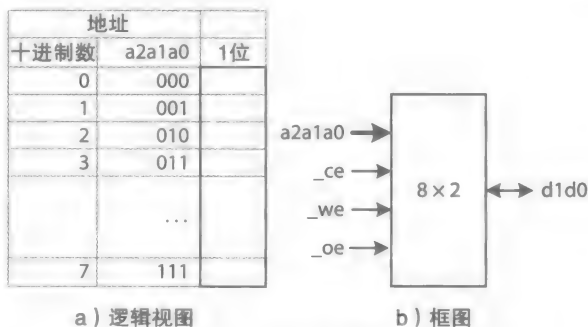
单元中的 1 位数据出现在数据线 d_0 、 d_1 、 d_2 、 d_3 上。激活列选择信号将使能某一列的三态缓冲门,使得数据线 d_0 、 d_1 、 d_2 或 d_3 上的数据输出到 d_{out} 上。在一个读操作过程中,当 $_we = 1$ 时,若 $_oe$ 被选中, d_{out} 中的数据将输出至数据总线上。其中单独的输出三态缓冲门是用来设计实现标记为 d 的 1 位位宽的双向数据总线。

存储器写操作过程与读操作过程相似,但此时,单输入三态缓冲器将被使能从而将输入数据 d 传送到相应的存储单元中。在读操作过程中,通过使用三态缓冲器降低了 SRAM 的功耗。 $_we$ 、 d 和列选通信号用于生成 $S_i = d$ 和 $r_i = \bar{d}$,这两个信号是用于激活行中的目标存储单元。该激活行中其他存储单元的 S_i 和 r_i 输入信号将为 0,使得其他存储单元保持之前的 1 位数据。

例 7-2 请设计一个 8×2 的 SRAM,要求 3 位宽的地址总线、2 位宽的双向数据总线以及 3 位宽的控制总线。

解: 图 7-11 展示了 8×2 SRAM 的逻辑视图和框图,SRAM 有 8 个地址,每个地址中可存储 2 位数据,图 7-12 展示说明了该 SRAM 的内部组织结构,其包含两个 $4 \times 2 \times 1$ 的存储单元阵列、一个 2-4 行译码器以及一个 1-2 列译码器。

8×2 SRAM 使用了与图 7-10 中 SRAM 相同的 4×4 存储单元阵列,但与之不同的是, 8×2 SRAM 中使用 1-2 列译码器对信号进行译码从而分别从两个 $4 \times 2 \times 1$ 单元阵列中各选择一个存储单元。此外,该存储器还包含两个输入三态缓冲门和两个输出三态缓冲门,2 位宽的双向数据总线信号线标记为 d_0 和 d_1 。



a) 逻辑视图

b) 框图

图 7-11 8×2 SRAM 逻辑视图和框图

7.4.4 存储单元设计

一般情况下,一个单独的存储芯片并没有足够的存储空间以支持一个如计算机一样复杂的数字系统。在系统执行过程中,处理器可同时操作多位数据(如 16 位、32 位、64 位),处理器可能需要多达几 GB 的存储空间来存储程序运行中的指令和数据,这些指令和数据都被看作存储数据。存储单元是指存储空间的结构,已知为可被一个或多个处理器访问的主存储器。当有一个以上的处理器,每个处理器必须轮流访问存储器单元,而存储器单元通常采用一个或多个存储模块设计(如存储卡),每个存储模块可存储多个字节数据,通常每个存储地址存储 4B 或 8B。

存储单元的内部组织结构是由存储数据在多个独立引用的存储空间内的分布方式所决定的。在一种组织结构中,一串连续地址的数据可能被存储在一个存储模块中,在另一种组织结构中,则可能被存储在多个不同的模块,当采用多片存储单元组织结构时,数据甚至可被存储在多个不同的片中。在 7.6 节中,我们将介绍多个不同的存储器存储数据的组织结构,在剩下的章节中,术语存储器和主存储器指的是一个或多个存储器单元。

1. 存储模块

一个存储卡的内存模块可以是单列直插式内存模块(SIMM),也可是双列直插式内存模块(DIMM),但单列直插式内存模块如今已不常用。在 SIMM 中,所有的管脚在存储卡一面上;DIMM 要比 SIMM 小,其管脚一半在一面,一半在另一面。图 7-13 说明了一个

16×8 的 SIMM 的示例, 其包含 8 个 16×1 的存储器芯片, 地址总线、数据总线和控制总线分别和 8 个存储器芯片相连接, 从而可满足同时的存储器读写访问请求, 图中的三态缓冲门用于避免未进行读写请求时从总线方向输入的违规信号。

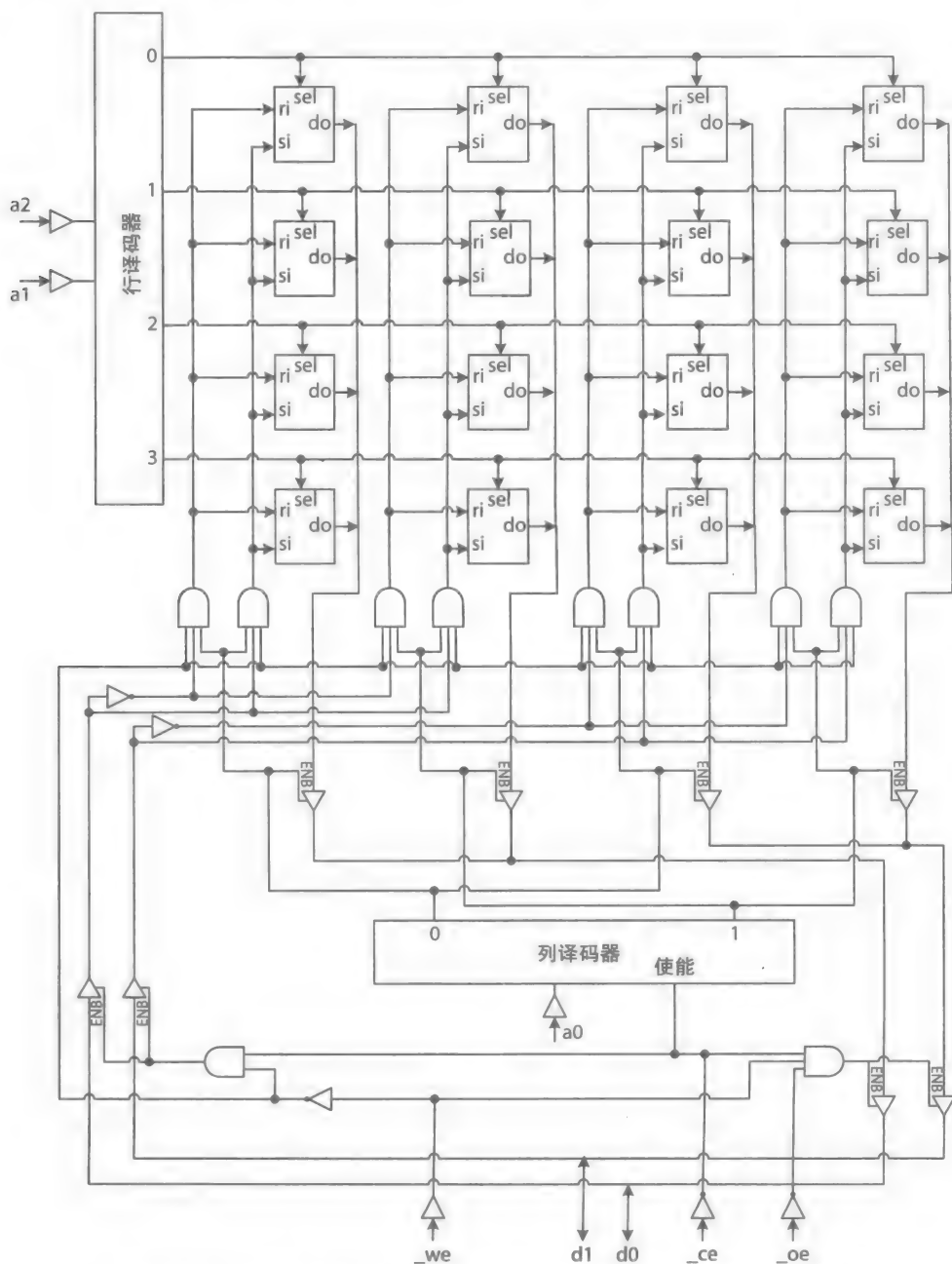


图 7-12 使用两个 4×2×1 单元阵列的 8×2 SRAM 内部组织结构

DIMM (SODIMM) 外形较小, 其大小约为普通 SDRAM DIMM 的一半。对于内存模块的其他示例, 请参阅 [5]。具有错误纠正编码 (ECC) 功能的 SDRAM 使用海明码来发现和纠正 1 位的内存错误。例如, 一个 64 位的 ECC SDRAM 存储模块使用 8 个奇偶校验位来校

验 64 位数据的正确性，生成一个 72 位的海明码。

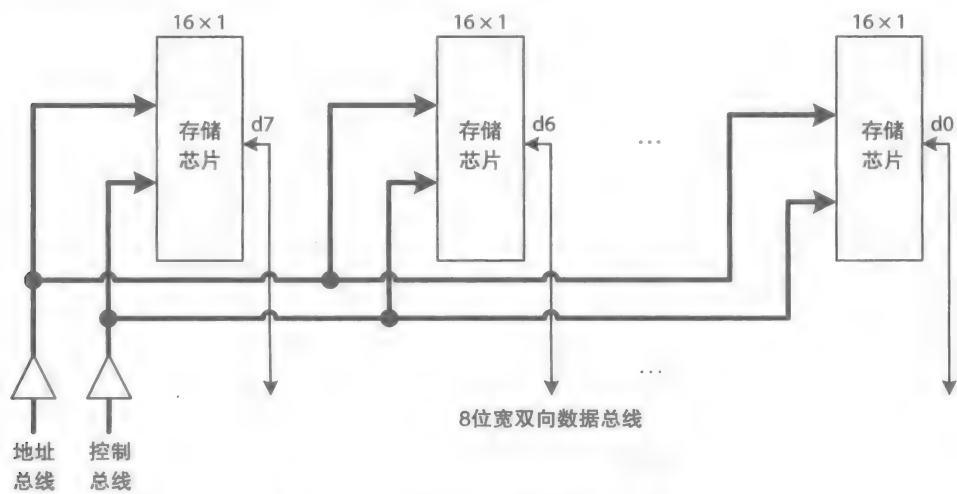


图 7-13 16×8 单列直插式内存模块

2. 存储单元

计算机通常有几个内存扩展槽供安装一个或多个内存卡，每个内存槽可安装一个内存卡。假设只有 2GB 和 4GB 两种 DIMM 可用，可用 4 个 2GB DIMM 或 2 个 4GB DIMM 来设计成一个 1G×64 的存储单元。具体采用哪种方式由计算机上可用的内存扩展槽的数量决定。

例 7-3 设计一个 64×4 (32B) 的存储单元，要求使用 16×4 的存储模块，存储模块使用 16×1 的存储芯片。

解：图 7-14 中展示了 64×4 存储单元的逻辑视图和框图，存储单元有 6 位地址信号线，标记为 a_5 到 a_0 ，有 4 位数据信号线，标记为 d_3 到 d_0 。存储单元需要 4 (64/16) 个存储模块来存储数据，每个存储模块占用 1/4 地址空间。6 位地址信号线中，有两位用于确定将要执行读写操作的目标存储模块，剩下的 4 位用于确定在已选择的存储模块中的 4 位的目标单元。例如，地址信号中的高两位 a_5 和 a_4 可将地址空间分割为 4 块，每块大小为 16，如图 7-14a 所示。在每一范围内的数据存储在对应的存储器模块中。

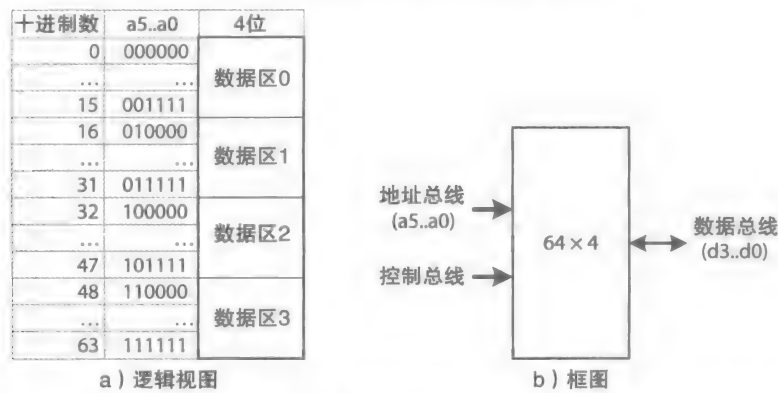


图 7-14 64×4 存储单元：a) 划分为 4 个数据区的逻辑视图；b) 框图

存储单元需使用一个 2-4 译码器来转换地址 a_5 和 a_4 为 4 个芯片使能信号 $_ce_0$ 到 $_ce_3$ ，

287
288

每个使能信号用于使能对应的存储模块，如图 7-15 所示，该译码器只能在主芯片被选中时（如 _ce ）才可被使能。地址信号 a_3 到 a_0 与剩余的控制总线信号（当存储器使用 SRAM 时，控制信号为 _we 、 _oe ，当使用 DRAM 或 SDRAM 时，控制信号为 _ras 、 _cas 、 _we 、 _oe ）将用于读写操作，只有使能的存储模块才可使用 4 位的数据总线传输数据。

图 7-15 展示了 64×4 存储器单元的内部组织结构，其将图 7-14a 中各范围内的数据存储在对应的存储模块中。其他数据存储结构将在 7.6 节中阐述。

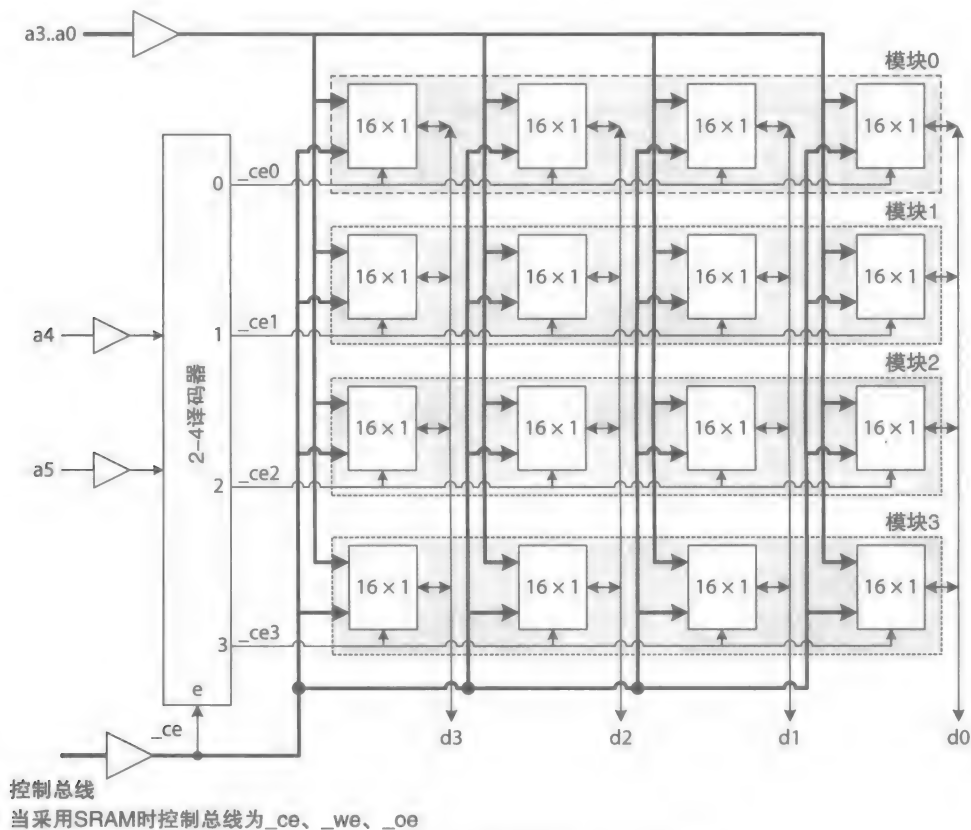


图 7-15 64×4 存储结构的 32B 存储器

7.5 存储时序

存储器的时序图精确地阐明了一个存储器的交互协议。它指定了 SRAM 或 DRAM 的控制信号以及 SDRAM 存储命令执行的时序过程。选中目标存储单元和执行完一个读写操作所需要的所有时间叫作**存储器存取时间**。一个**存储周期**包括存取时间和数据传输时间两部分。

存储器的存取时间和存储单元阵列的大小直接成正比，存储单元的大小决定了存储器芯片中行译码器、列译码器的大小，而两个译码器的大小决定了存储器激活一行和存取目标单元所需要的时间。此外，因为译码器是多级的，可能有更长的传输时延。虽然存储器单元、模块、芯片之间的交互协议是相同的，但是采用不同存储技术的存储器之间的交互协议并不相同。

7.5.1 SRAM

图 7-16 从存储器视角说明了一个 SRAM 读操作周期，我们也可从 CPU 的视角来绘制

存储器的时序图，关于 CPU 视角的时序图将在第 9 章讨论。

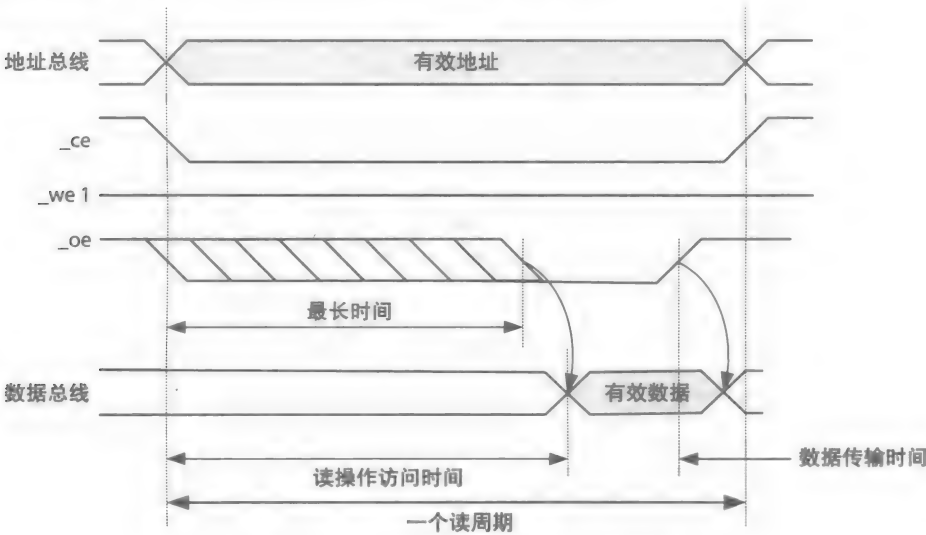


图 7-16 存储器视角的 SRAM 读周期

在存储器读周期中，存储地址将首先放置在地址总线上，同时或稍后存储器 `_ce` 信号被选中。
`_oe` 信号和 `_ce` 信号共同控制一个或多个输出三态缓冲门（如图 7-12）。为了最小化存储器读操作周期时间，`_oe` 信号可在 `_ce` 信号被选中后的最长时间（maximum time）范围内的任意时间被选中，如时序图中所示。当存储单元阵列中的数据可用时，`_oe` 信号使能，从而允许数据输出到数据总线上；因为 `_ce` 信号与输出三态缓冲器和列译码器相关，将在最后解使能，从而关闭输出三态缓冲门和列译码器。

存储器的写周期和读周期相似，但有不同之处，需存储的数据必须要在 `_ce` 信号选中同时或者在 `_we` 信号选中后的最长时间内放置到数据总线上，使得占用数据总线时间最少。图 7-17 说明了 SRAM 写周期时序过程。

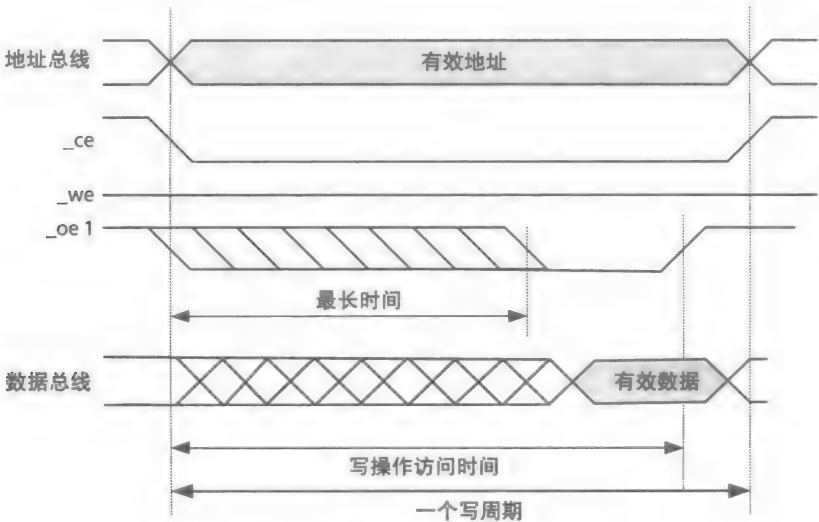


图 7-17 存储器视角的 SRAM 写周期

存储器的存储周期由 CPU 启动，通常需要多个 CPU 时钟周期来完成。

7.5.2 DRAM

在 DRAM 的读写周期中，目标存储单元地址需要被分割为行地址和列地址两部分，通过地址总线传送给存储器。这降低了地址总线的宽度，并且降低了完成一次突发访问所需要的所有时间。当进行短突发数据访问并且一行已激活时，多个列地址将顺序执行从而快速完成多个数据的读写操作。图 7-18 从存储器视角说明了 DRAM 读周期和写周期。

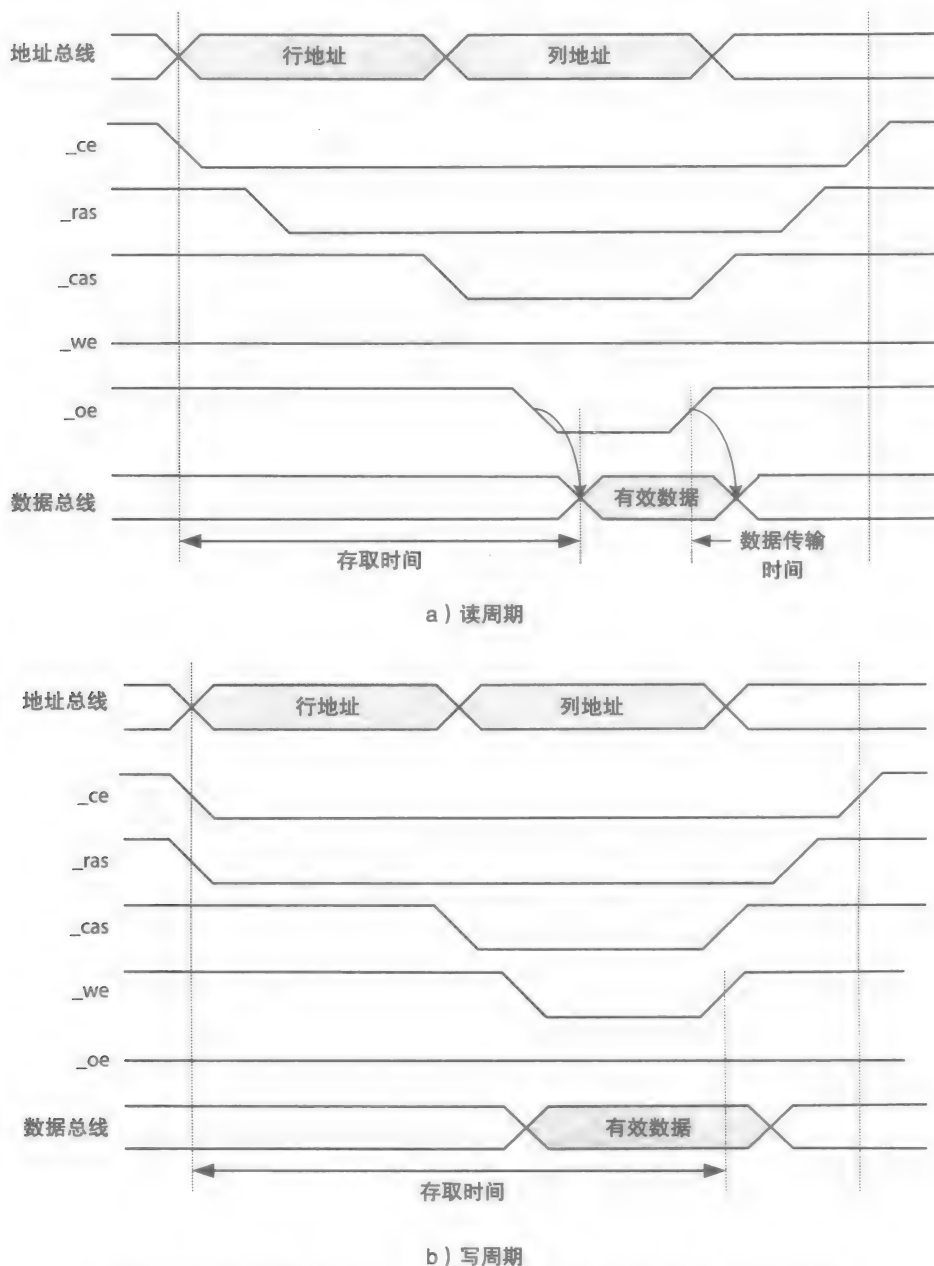


图 7-18 存储器视角的 DRAM 读 / 写周期：a) 读周期；b) 写周期

DRAM 的读写周期从选中 $_ce$ 信号和输入行地址开始。接下来, $_ras$ 信号将被选中从而使得 DRAM 可加载行地址到一个内部寄存器中并激活该行。然后输入列地址并选中 $_cas$, 这将选中一个或多个位于激活行上的目标单元, $_oe$ 信号和 $_we$ 信号的功能与之前 SRAM 中描述的 $_we$ 、 $_oe$ 相同。

DRAM 除了读写周期之外, 还需要一个刷新周期来刷新每个存储单元中的内容。列先于行的刷新周期要求 $_cas$ 信号选中在 $_ras$ 信号选中之前进行, 从而将 DRAM 切换为刷新模式。DRAM 用于构建采用了具有简化计算机特性的标准通信协议的 SDRAM, 因此, 这也有助于降低计算机的成本。

7.5.3 SDRAM

SDRAM 除了采用的标准通信协议之外, 还实现了不同的刷新周期, 包括正常模式和部分省电模式 [3]。一个 SDRAM 存储周期从一条行激活命令开始, 之后有一条或多条针对该行存储单元的读写命令。

图 7-19 说明了在假设突发访问大小为 4 时 (大小为 4 指的是每次突发访问存取 4 个存储单元的数据) 进行两个连续读周期的 SDRAM 时序图, 图中假定一个 SDRAM 总线时钟周期可读入一个突发访问大小, 需要 3 个时钟周期来激活一行, 需要 2 个时钟周期来完成存取操作, 每个数据需要一个时钟周期从 SDRAM 输出。此外, 因为 SDRAM 采用管道化的数据通路, 可以在前一个数据传输出 SDRAM 的同时开始同一行的下一个读周期。

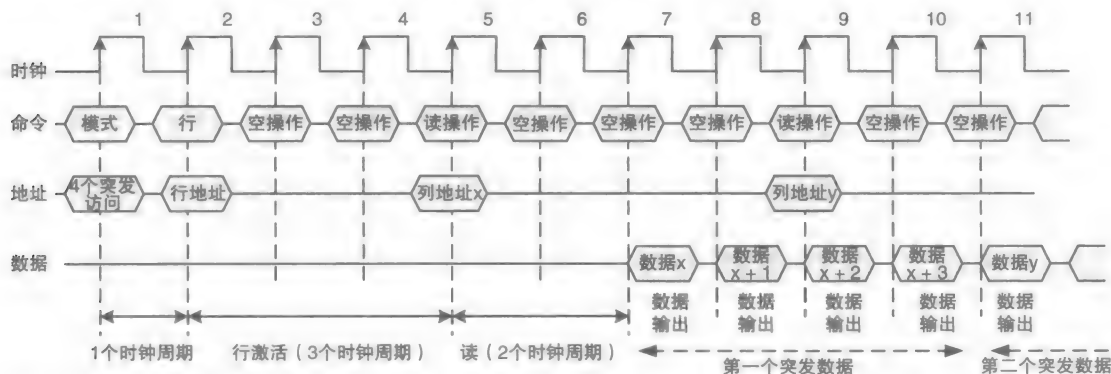


图 7-19 进行两个连续读周期的 4 位突发访问的 SDRAM 时序图, 设定一个 SDRAM 总线时钟周期可读入一个突发访问, 需要 3 个时钟周期来激活一行, 需要 2 个时钟周期来完成存取操作

特别的, 在图 7-19 中, 4 位的突发访问在第 1 个时钟周期就出现在总线上 (突发访问数据的类型未说明), 在第 2 个时钟周期, 执行一条行激活命令, 行激活命令需要行地址来明确激活的是哪一行。在第 4 个时钟周期末, 该行被激活。第 5 个时钟周期, 地址总线输入列地址 x 并执行读命令, 该行中列地址 x 到 $x+3$ 中的数据将从第 7 个时钟周期及其之后的 3 个时钟周期依次出现在数据总线上, 列地址 $x+1$ 到 $x+3$ 是由存储器内部自动生成的。第 2 个读命令 (从同一行读取) 是在第 9 个时钟周期执行, 其 4 个突发数据将从第 11 个时钟周期开始通过列地址 y 到 $y+3$ 输出到数据总线上。图中仅绘制了地址 y 中数据的读取过程。

图 7-20 说明了在突发访问大小为 1 时, 在同一行中进行 2 个连续读周期再进行 2 个连续写周期的 SDRAM 时序图。因为 SDRAM 的数据通路是管道化的, 在突发访问大小为 1 时执行连续的读命令和写命令可以每个时钟周期执行一次。

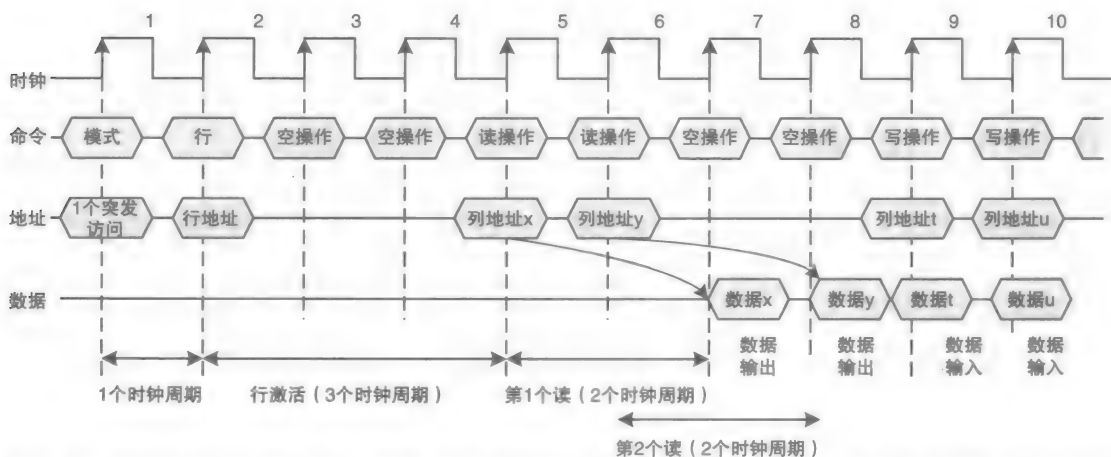


图 7-20 突发访问大小为 1 时, 在同一行中进行 2 个连续读周期再进行 2 个连续写周期的 SDRAM 时序图, 假定每个时钟周期进入一个突发访问, 需 3 个时钟周期进行行激活, 2 个时钟周期完成读操作或写操作

7.5.4 DDR SDRAM

DDR SDRAM 对 SDRAM 进行了优化, 从而使得其有效带宽达到了 SDRAM 的两倍。在 DDR SDRAM 中, 可在每个时钟的上升沿和下降沿进行数据传输。图 7-21 说明了对同一行进行大小为 4 的突发访问时连续 3 个读操作的 DDR SDRAM 时序图。如图所示, 在 SDRAM 中, 一个读周期内的 4 个数据在两个时钟周期内传输完成, 而所有的 12 个数据在 6 个时钟周期内完成传输。因此, DDR 有效提高了存储器峰值带宽一倍, 其中, 存储器峰值带宽指的是在存储器中数据总线被 100% 使用时的最大存储器带宽。

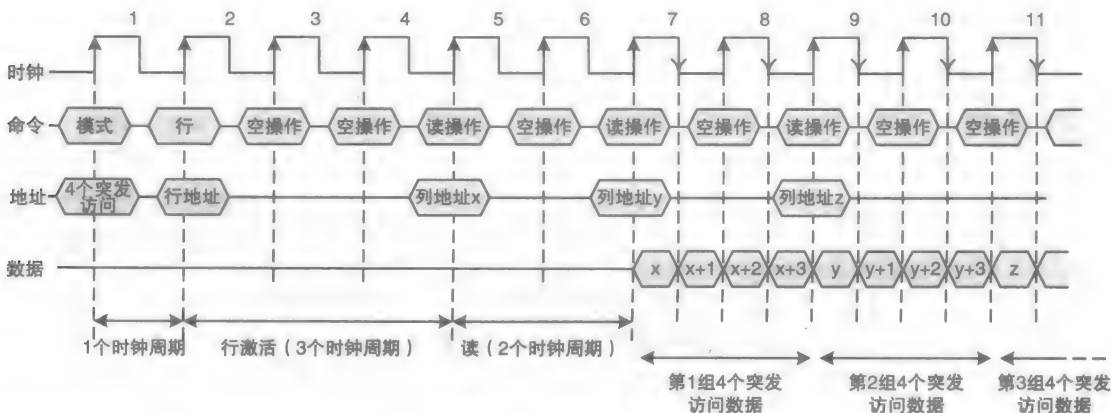


图 7-21 突发访问大小为 4 时, 在同一行中进行连续 3 个读周期的 DDR SDRAM 时序图, 假定每个时钟周期进入一个突发访问, 需 3 个时钟周期进行行激活, 2 个时钟周期完成读操作或写操作

7.6 存储器体系结构

存储器体系结构指的是数据以多种方式存储在 (分布在) 两个或更多的存储器单元, 两个或更多个存储器模块, 或者两个或更多个存储芯片 (单元阵列), 从而提升存储器效率并

实现更高的带宽。例如，在每个处理器与其专用存储器单元连通的双处理器系统中，如果每个处理器访问的数据存储在分配给该处理器的专用存储器单元中，存储系统将更加高效。此外，存储在各个存储器单元中的数据可以以多种方式组织从而优化并提升存储器带宽，进而满足处理器中的处理器核心对数据速率的要求。

7.6.1 高位交叉存储

高位交叉存储技术是将整个逻辑存储空间分割为两个或多个连续的数据区，每个数据区中的数据存储在独立分开的存储模块或存储单元中。例如，考虑 1GB 大小的两路高位交叉存储数据被存储在两个存储单元中，如图 7-22 所示，在这个例子中，数据区是通过数据的字节地址来分割的，分成字节地址 0 ~ 512M - 1（高一半，从地址 0 开始）和 512M ~ 1G - 1（低一半）。

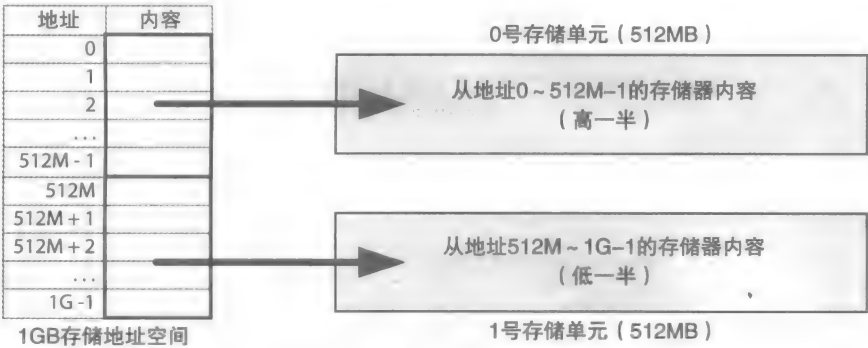


图 7-22 采用 2 路高位交叉存储方式的 1GB 数据

在一个 k 路高位交叉存储中，使用 m 个最高地址位将逻辑存储空间划分为 2^m ($2^m \geq k$) 个区域。以图 7-15 中的存储器组织为例，在图中， 64×4 存储单元组织结构的 32B 存储空间采用 4 路高位交叉存储方式，划分为 4 个区域，依次为字节地址 0 ~ 15（区域 0）、16 ~ 31（区域 1）、32 ~ 47（区域 2）、48 ~ 63（区域 3）。正如图中所示，每个区域中的数据被存储在对应的独立分开的存储模块中。

7.6.2 低位交叉存储

考虑一个多存储片存储器组织结构。低位交叉存储，通常也被叫作简单交叉存储，将第一个数据存储在一个存储片中，将下一个顺序存取数据放在其他存储片中，依此类推，周而复始。例如，假设每一个存储的数据是一个 4B 的字，使用 4 路低位交叉存储方式，存储字 0 到存储片 0 中，存储字 1 到存储片 1 中，存储字 2 到存储片 2 中，存储字 3 到存储片 3 中，周而复始，存储字 4 到存储片 0 中，字 5 到存储片 1 中，依此类推。图 7-23 说明了一个采用 4 路低位交叉存储方式的组织成 128M × 4B 结构的 512MB 存储空间，在这个示例中，每 4 个连续的 4B 字存储在 4 个不同的存储片中。

295

图 7-24 说明了采用低位交叉数据存储方式的两片 DDR SDRAM 时序图，在这个示例中，相同时间内数据存储的数量从图 7-21 中的 4 个变为图 7-24 中的 8 个，在没有改变 SDRAM 内部时钟频率的情况下，有效提升了存储器有效带宽 1 倍。2 倍的数据量必须使用 2 倍位宽的数据总线或 2 倍频率的时钟来完成数据的传输。这是 DDR2、DDR3 等 SDRAM

的设计中,访问的数据倍增的原因。

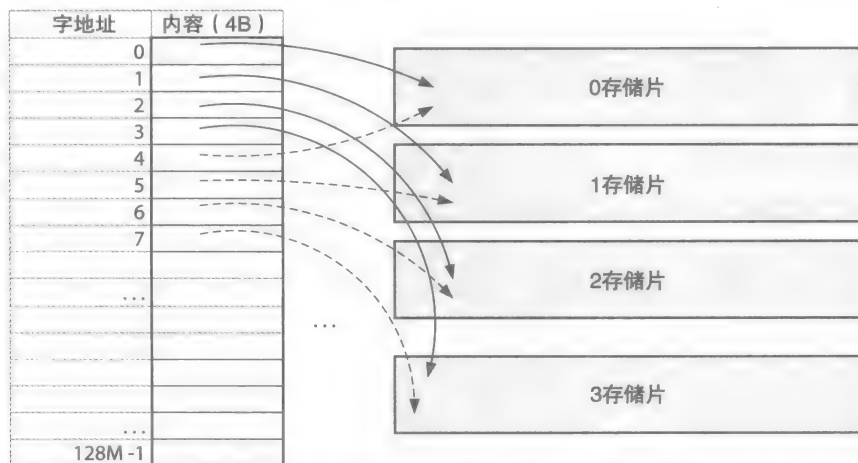


图 7-23 采用 4 路低位交叉存储方式的 512MB 数据存储于 4 个存储片中

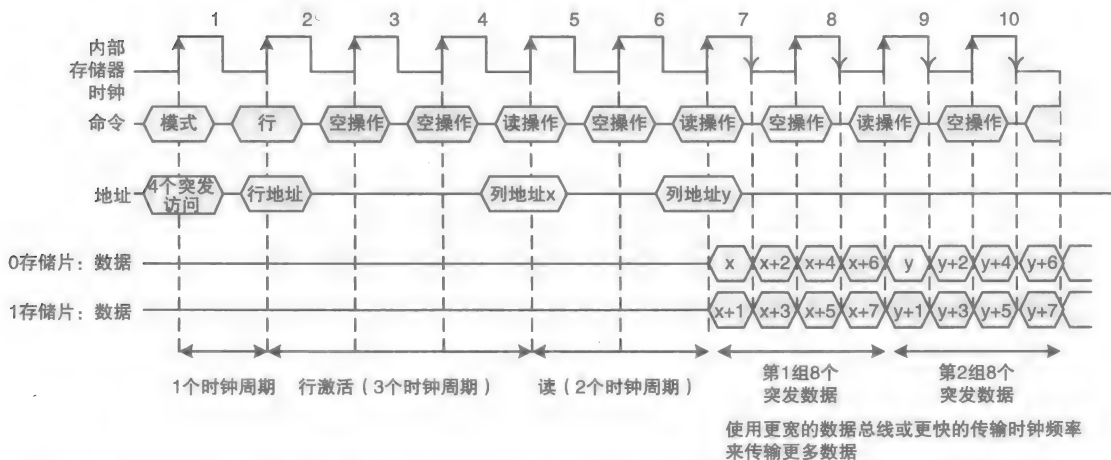


图 7-24 采用低位交叉数据存储方式的两片 DDR SDRAM 时序图, 假定一个时钟周期进入一个突发访问, 需 3 个时钟周期激活一行, 2 个时钟周期完成读写访问

低位交叉存储方式也被用于存储模块中来进一步提升存储器带宽。例如, 考虑一个具有 2 个存储模块、采用 2 位低位交叉存储方式的存储单元, 此外, 假定每个存储模块采用 2 个存储片的 DDR SDRAM (如 DDR2) 芯片。在这个示例中, 存储单元可传输的数据是图 7-24 中的 2 倍。这意味着, 每个 SDRAM 时钟周期中每个存储模块可传输 4 个数据, 2 个在上升沿 (x 和 $x+1$), 2 个在下降沿 ($x+2$ 和 $x+3$)。存储单元将每个时钟周期内传输 8 个数据 (每个存储模块传输 4 个), 因此每个存储模块的带宽是之前的两倍。

7.6.3 多通道

多通道存储器是指存储器架构中存储器与系统其他部分有两个或多个通信信道 [6, 7]。例如, 图 7-25 说明了一个具有 CPU 和数字信号处理器 (DSP) 的双系统与一个单通道或双通道的存储器通信的架构。包含一系列总线或点对点连线 (将在第 9 章讨论) 的互联网

络被用于连接 CPU、DSP 和存储器，在这个例子中，每个通道可能用于服务不同的处理器单元。

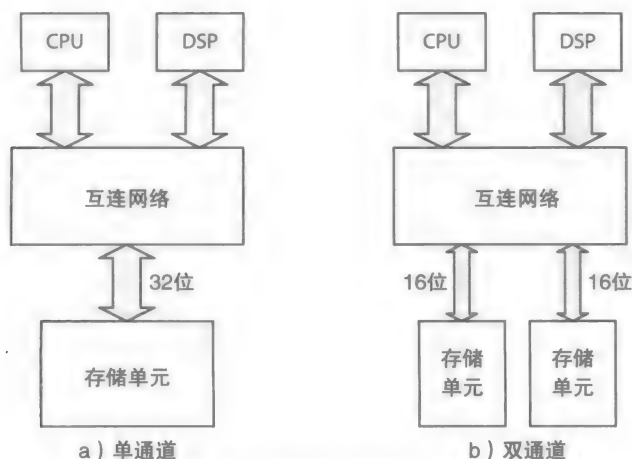


图 7-25 单通道存储器架构和双通道存储器架构

在一个单通道体系结构中，CPU 和 DSP 必须交替访问存储器，而在一个双通道体系结构中，两个处理器可同时访问存储器，其中每个处理器通过不同的通道访问存储器，条件是每个处理器需要的数据存储在不同的存储单元中。

一个多通道存储单元，如果以更大的突发访问大小来存取同样的数据，存储效率将更高。换句话说，在大多数数据并不是向一个单独的存储模块存储的情况下，传输更大的突发访问时，每个通道都可用于连续传输数据从而提升存储器效率。这使得多通道存储器架构更加适合需要连续传输数据的实时系统。

例 7-4 考虑一个 64 位单通道和一个 32 位双通道存储器结构，如图 7-25 所示。假定存储器单元设计使用 DDR SDRAM 模块。我们需确定当 CPU 在同一行中使用 4 个独立的突发读周期时访问 128B 数据时每个通道的效率，其中每次读 32B 数据。

解：公式 (7-1) 说明了存储器的效率

$$\text{存储器效率 (ME)} = \frac{\text{传输数据使用的数据总线时钟周期数}}{\text{所有数据总线时钟周期数}} \quad (7-1)$$

为了从单通道存储器中访问 32B 的数据，突发访问的大小必须是 4 (32B/64b)，如图 7-21 所示。在第一个数据出现在数据总线上之前需要 6 个时钟周期，每读 32B 数据需要 2 个周期，共需要 8 (2×4) 个周期区访问 128B (4×32B) 数据，因此，存储器效率为 57%，计算如下：

$$ME_{\text{single-channel}} = \frac{8 \text{ 个时钟周期}}{(6 + 8) \text{ 个时钟周期}} = 0.57 \text{ 或 } 57\%$$

另一方面，为了使用双通道存储器中的一个通道来访问 32B 数据，突发访问的大小必须为 8 (32B/32b)。正如在一个单通道存储系统中，在第一个数据出现在数据总线上之前也需要 6 个时钟周期，每读 32B 数据需要 4 个时钟周期，访问 128B 数据共需要 16 (4×4) 时钟周期，在这个示例中，通道效率增加到 73%，计算如下：

$$ME_{\text{each-channel}} = \frac{16 \text{ 个数据时钟周期}}{(6 + 16) \text{ 总时钟周期}} = 0.727 \text{ 或 } 73\%$$

7.7 设计实例：多处理器存储结构

正如前面所讨论的，一方面，低位交叉存储通过并行访问多个存储片或多个存储模块从而增加存储器带宽。另一方面，高位交叉存储用于分割存储空间为多个区域并存储每个区域中的数据到独立分开的存储模块或存储单元中。

7.7.1 UMA 与 NUMA

图 7-26a 中展示了一个不统一存储器访问（NUMA）多处理器系统体系结构。在这个示例中，每个处理器都分配了单独的存储单元，创建一个处理节点。一个互连网络连接了各个处理节点，生成一个共享的存储器系统。

在 NUMA 系统中，每个处理器与它的本地存储单元（简单的本地存储器）距离更短，而与他远程存储器（另一个处理器的本地存储器）距离更长。因此，一个存储器读写周期的时延，即存储器延迟，在存储器从本地存储器中访问时比正常情况要短；而当存储器从远程存储器中访问时存储器延迟要比正常情况更长。因此，在一个 NUMA 系统中的读写存储器周期是不一致的，有时候短，短到几个时钟周期，有时候长，长到需要许多时钟周期。

另一方面，在一个统一存储器访问（UMA）结构的多处理器系统中（图 7-26b），所有的处理器以多通道的方式访问一个单独的存储单元。与 NUMA 存储器系统不同的是，不同的处理器在访问存储器时，存储器延迟没有太大变化，存储器延迟基本相同（平均统一的）。而且，在一个包含 NUMA 和 UMA 的系统中，存储单元可以采用低位交叉存储方式从而更好地服务在每个处理器中的处理器核心。

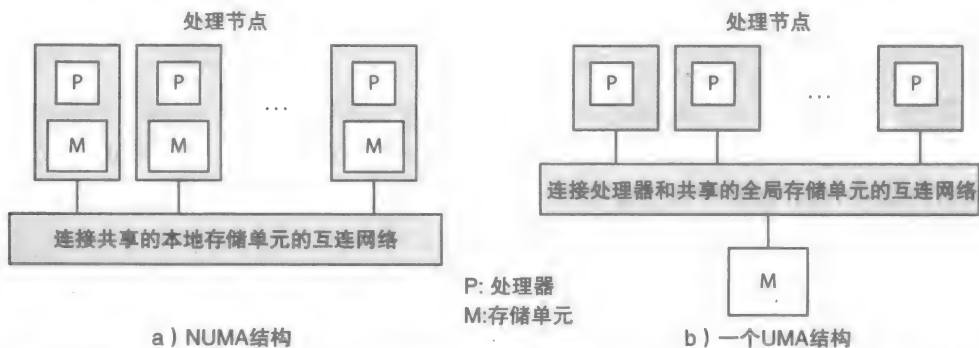


图 7-26 NUMA 和 UMA 系统结构：a) 使用共享的但本地分配存储单元的 NUMA 系统结构；b) 使用共享的全局存储单元的 UMA 系统结构

例如，美国硅图公司的 Altix 4700 系统是一个 NUMA 系统，可支持 512 ~ 1024 个处理器，并支持高达 128TB 的共享存储器 [8]。然而，今天常用的小型系统的架构多是 NUMA，比如使用双核处理器的 AMD Quad FX 平台 [9]。

7.7.2 NUMA 应用

在一个 NUMA 系统中，当使用两个或多个处理器协作处理完成一个任务时，低位交叉存储方式也可用于两个或多个存储单元，称为存储器节点交叉存储。例如，如果有两个处理器协作执行一个图像处理任务，其中一个处理器产生中间结果供第二个处理器处理，这将有

利于第一个处理器(P0)从他的本地存储单元(M0)中读取图像数据,但不利于第二个处理器(P1)写入中间结果给其存储器单元(M1)。这一矛盾可以通过创建结构来解决,例如创建一个结构数组,每个结构中包含2个数据。例如,考虑下面的程序代码,代码中定义了一个样例数据结构命名为“foo_t”,该结构体中包含元素a和元素b。假定P0操作元素a,生成元素b,P1修改元素b。使用存储器节点交叉存储方式时,所有的元素a将被存储在M0中,而元素b将被存储在M1中。

```
typedef struct{
    int a;
    int b;
}foo_t;
Main()
{
    foo_t array[100][100];
    ...
}
```

在这个示例中,P0用更短的时延来从其本地存储单元(M0)中读取图像元素a,用更长的时延来将计算好的图像数据结果b存储到P1的本地存储器单元(M1)中,之后P1将用更少的时延来从其本地存储单元中读取图像数据b。[我们将在第10章看到,P1可以花销更长的时延从P0的高速缓冲存储器(cache)中访问P0刚计算出的元素b。然而,为了简单起见,假定了P0用更长的时延将b全部写入M1中,P1使用更短的时延从M1中读取全部的计算结果b。关于时延的更精确的分析需要一个程序执行的仿真环境。此外,由于使用了高速缓冲存储器,这很可能花销更长的存储器时延来写入全部计算好的数据结果b到M1中,大多数情况下,程序将在后台执行而平均存储器时延不会增加。]一旦P0计算出多个图像结果b时,P0和P1将并行操作(在同一时间),将花销更少的时延来从它们各自的本地存储单元中访问图像数据。因此,这将导致在NUMA系统中程序执行时间的减少。另一方面,在一个等价的UMA系统中,P0和P1都将访问同一个存储单元,对于这一程序的存储器平均时延将相应的延长,继而将增加程序在UMA系统中执行的总时间。

300

7.8 HDL 模型

例7-5描述了一个存储单元的行为级和结构级组合的模型,存储模块使用SRAM芯片建模。存储单元使用高位交叉存储方式,与图7-15中展示的相似。SRAM芯片HDL模型使用双向数据总线,并使用一个简化的存储器交互协议,这指的是存储器读/写控制信号,当选中或未选中时,都是同时进行的。关于协议的更精确的模型需要使用更精确的时序来生成存储器控制信号,正如图7-16和图7-17中所示。这将要求存储器芯片(图7-10)使用原理图设计工具建模或者使用包含时序信号延时的HDL模型(结构的或行为的)。

例 7-5 使用4个 16×8 存储模块设计并仿真一个 64×8 存储单元, 16×8 的存储模块是由2个有双向数据线的 16×4 SRAM芯片构成。使用关键字“inout”来声明变量data为双向数据线。注意在SRAM芯片模型中,当存储器不处于读模式时,赋值语句将使得data高阻抗(Z)。当向存储单元中写数据时($_ce=0$, $_we=0$, $_oe=1$),data将作为输入,当从存储单元中读数据时($_ce=0$, $_we=1$, $_oe=0$),data将通过4位的地址线adrs来设置为存储器中的内容。

HDL 模型:

```

`include "ram16x8.v"
module ram64x8(
    input [5:0] adrs,
    inout [7:0] data,
    input _ce, _we, _oe
);
    reg [3:0] _cee;
    ram16x8 u1(adrs[3:0], data, _cee[0], _we, _oe);
    ram16x8 u2(adrs[3:0], data, _cee[1], _we, _oe);
    ram16x8 u3(adrs[3:0], data, _cee[2], _we, _oe);
    ram16x8 u4(adrs[3:0], data, _cee[3], _we, _oe);
    //2-to-4 decode
    always@(*)
    begin
        if(_ce == 0)
            case(adrs[5:4])
                0: _cee = 4'b1110;
                1: _cee = 4'b1101;
                2: _cee = 4'b1011;
                3: _cee = 4'b0111;
                default: _cee = 4'hf;
            endcase
        else
            _cee = 4'hf;
    end
endmodule

`include "ram16x4.v"
module ram16x8(
    input [3:0] adrs,
    inout [7:0] data,
    input _ce, _we, _oe
);
    ram16x4 u1(adrs, data[7:4], _ce, _we, _oe);
    ram16x4 u2(adrs, data[3:0], _ce, _we, _oe);
endmodule

module ram16x4(
    input [3:0] adrs,
    inout [3:0] data,
    input _ce, _we, _oe
);
    reg [0:15][3:0] mem; // 16 X 4 RAM
    assign data = ~_ce & _we & ~_oe ? mem[adrs]: 4'hz;
    always@(*)
    begin
        if(_ce == 0)
            if(_we == 0 && _oe == 1)
                mem[adrs] = data;
    end

```

```
end
endmodule
```

仿真测试平台：

仿真测试模型已经给出，在这个模型中，当存储单元在写操作（_ce=0, _we=0, _oe=1）过程中，assign 语句将用来将命名为 content 的 8 位数据放置到双向数据总线上。仿真了 4 种测试样例：先进行两个存储器写周期再进行两个存储器读周期。测试向量仿真了简化的存储器读 / 写周期。写周期从一个目标地址 adrs、一个数据变量 content 以及控制写操作信号的使能开始，写控制信号将在写周期的末尾进行解使能。读周期从目标地址 adrs、读控制信号的使能开始，读控制信号将在读周期末尾解使能。

302

这类似于 CPU 在执行中加载、存储指令时访问存储器。写周期以执行一个存储指令为开始，存储指令将提供一个存储器地址和一个将存储到存储器的值（如寄存器内容）。读周期以执行一个加载指令为开始，加载指令将提供一个存储器地址并存储从存储器中取出的值到一个寄存器中。因为 CPU 也会执行算术指令以及其他类型的指令，故而 CPU 不会一直访问存储器，在测试平台中，这将通过在存储器读 / 写周期之间添加随意大小的延时来仿真实现。

```
'include "ram64x8.v"
module ram64x8test();
reg [5:0] adrs;
reg [7:0] content;
reg _ce, _we, _oe;
wire [7:0] data;
assign data = ~_ce & ~_we & _oe ? content : 8'hz;
ram64x8 ul(adrs, data, _ce, _we, _oe);
initial begin
$monitor ("%4d: adrs = %h _ce = %b _we = %b _oe = %b data = %h", $time, adrs, _ce, _we, _oe, data);
//two memory writes
adrs = 6'b001111; //memory module 0
content = 8'haa; //data as content
_ce = 0; _we = 0; _oe = 1; //begin a write cycle
#10
_ce = 1; _we = 1; _oe = 1; //end a write cycle
#50 //some delay
adrs = 6'b011111; //memory module 1
content = 8'hbb;
_ce = 0; _we = 0; _oe = 1;
#10
_ce = 1; _we = 1; _oe = 1;
#40
//two memory reads
adrs = 6'b001111;
_ce = 0; _we = 1; _oe = 0; //begin a read cycle
#10
_ce = 1; _we = 1; _oe = 1; //end a read cycle
#60
adrs = 6'b011111;
_ce = 0; _we = 1; _oe = 0;
```

303

```
#10
    _ce = 1; _we = 1; _oe = 1;
#30
$finish;
end
endmodule
```

仿真结果:

正如所预期的那样, 命名为 data 的存储器数据线在存储器不被访问时将变为高阻抗 (Z) 状态。

```
Chronologic VCS simulator copyright 1991-2009
Contains Synopsys proprietary information.
Compiler version D-2009.12; Runtime version D-2009.12; May
20 22:54 2014
    0: adrs = 0f  _ce = 0 _we = 0 _oe = 1 data = aa
   10: adrs = 0f  _ce = 1 _we = 1 _oe = 1 data = zz
   60: adrs = 1f  _ce = 0 _we = 0 _oe = 1 data = bb
   70: adrs = 1f  _ce = 1 _we = 1 _oe = 1 data = zz
  110: adrs = 0f  _ce = 0 _we = 1 _oe = 0 data = aa
  120: adrs = 0f  _ce = 1 _we = 1 _oe = 1 data = zz
  180: adrs = 1f  _ce = 0 _we = 1 _oe = 0 data = bb
  190: adrs = 1f  _ce = 1 _we = 1 _oe = 1 data = zz
$finish called from file "ram64x8test.v", line 41.
$finish at simulation time                220
```

参考文献

1. Flash Memory: An Overview, <http://www.spansion.com/Support/TechnicalDocuments/Pages/TechnicalDocuments.aspx>.
2. C. R. Nave, HyperPhysics, Georgia State University, <http://hyperphysics.phy-astr.gsu.edu/>.
3. Memory chips from Micron, <http://www.micron.com/products/dram/>.
4. Memory from Rambus, <http://www.rambus.com/>.
5. Memory modules, <http://www.newegg.com/>.
6. Gomony MD, Akesson B, Goossens K. Architecture and optimal configuration of a real-time multi-channel memory controller, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, 1307-1312.
7. Siqueira HM, Silva IS, Kreutz ME, Correa EF. DDR SDRAM memory controller for digital TV decoders, *Symposium on Computing System Engineering (SBESC)*, 2011, 78-82.
8. NUMA SGI Altix 4700 system, <http://www.sgi.com/products/>.
9. AMD Quad FX Platform, <http://support.amd.com/>.

练习

- 7.1 在下列两种情况下, 绘制一个 128KB 存储器的逻辑视图和框图, 并说明存储单元数量:
 - a. 地址总线宽度为 1B
 - b. 地址总线宽度为 2B
- 7.2 在下列存储器大小下, 分别说明其存储单元阵列结构
 - a. 128×1
 - b. 64×2

c. 32×4

- 7.3 使用图 7-8 中的 SRAM 单元模型, 设计一个 32×2 的 SRAM (不用画出每个存储单元)。
- 7.4 使用图 7-8 中的单元模型, 设计一个 16×4 的 SRAM (不用画出每个存储单元)。
- 7.5 设计一个 256B 的存储器, 要求其组织结构为 128×16 , 使用 32×4 的 SRAM 芯片作为存储模块。
- 7.6 设计一个 256B 的存储器, 要求其组织结构为 128×16 , 使用 16×8 的 SRAM 芯片作为存储模块。
- 7.7 设计一个 256B 的存储器, 要求其组织结构为 128×16 , 存储器划分为位于低地址的 128B ROM 空间和位于高地址的 128B RAM 空间, 设计存储器时使用 64×8 的 ROM 芯片和 32×8 的 SRAM 芯片。
- 7.8 研究以下存储器技术并以此写一篇小论文
- a. Rambus RDRAM
 - b. Rambus XDR
 - c. Rambus XDR2
 - d. EDO DRAM
- 7.9 绘制一个 SDRAM 的时序图, 其中先进行一个读周期再进行一个写周期, 读写周期的突发访问大小为 4。假定需要 1 个时钟周期来进入 1 个大小的突发访问, 需 4 个时钟周期来激活一行, 需要 3 个时钟周期来完成一次读或写操作。
- 7.10 一个数据总线宽度为 32 位的 SDRAM, 总线的时钟频率为 200MHz, 存储器的峰值带宽为多少 MB/s?
- 7.11 一个数据总线宽度为 64 位的 SDRAM, 总线的时钟频率为 200MHz, 存储器的峰值带宽为多少 MB/s?
- 7.12 一个数据总线宽度为 32 位的 DDR SDRAM, 总线的时钟频率为 200MHz, 存储器的峰值带宽为多少 MB/s?
- 7.13 图 7-19 中的 SDRAM 时序图中, 假设有 4 个存储器读周期, 顺序如下, 其中数据总线宽度为 32 位:
- 行地址 x , 突发大小为 4, 列地址 x_1 和 x_2 (共 32B)
行地址 y , 突发大小为 4, 列地址 y_1 和 y_2 (共 32B)
- a. 绘制其时序图。
 - b. 计算存储器效率, 忽略行解激活消耗的时间。
- 7.14 图 7-21 中的 DDR SDRAM 时序图中, 假设有 4 个存储器读周期, 顺序如下, 其中数据总线宽度为 32 位:
- 行地址 x , 突发大小为 4, 列地址 x_1 和 x_2 (共 32B)
行地址 y , 突发大小为 4, 列地址 y_1 和 y_2 (共 32B)
- a. 绘制其时序图。
 - b. 计算存储器效率, 忽略行解激活消耗的时间。
- 7.15 一个 64B 的存储单元, 由 16×8 的存储模块构成, 存储模块间采用高位交叉存储方式, 在每个存储模块中, 数据采用低位交叉存储方式。假设每个存储模块由 8 个 SDRAM 芯片, 每个 SDRAM 芯片有 4 个存储片并组织成 $2 \times 2 \times 1$ 的单元阵列。请描述一个 2 位的突发访问数据是如何存储到存储单元中的。
- 7.16 一个采用 400MHz 的 DDR SDRAM 存储模块的 4 通道存储器, 当每个通道为 64 位 (8B) 宽时, 存储器的峰值带宽为多少?

- 7.17 一个 NUMA 系统，其本地存储器的延时为 1τ ，远程存储器的延时为 4τ 。如果程序执行中 80% 的存储器访问在本地而 20% 的访问在远程存储器时，存储器延时的均值为多少？将结果与一个平均存储器延时为 4τ 的 UMA 系统进行比较。
- 7.18 在 128×32 的存储单元中，采用存储器节点交叉存储方式，请描述类型为 “foot_t” 的阵列 `array[8][8]`，其元素是如何物理的存储在存储单元 M0 和 M1 中的。

计算机安全

- 7.19 计算机安全（存储器认证）：选择练习 11.24 和 / 或练习 11.25（参看 11.9 节）。注意，关于高速缓冲存储器的细节在第 10 章中。然而，在这，首先确定存储器块的数量，然后分配每个块的块地址，块地址从地址 0 开始。

指令集体系结构

8.1 简介

前面的章节涵盖数字化设计的概念、技术以及存储器的组织和架构。在这一章中，我们将讨论关于 CPU 数据通路的相关内容。除非明确说明，本章中的术语 CPU 和处理器都指的是单核处理器，关于多核处理器的内容将在第 10 章讨论。

指令集体系结构 (ISA) 指的是执行程序的一个单周期、多个周期或流水线的数据通路。在这种情况下，数据通路能够承载许多不同指令的执行，每条指令都需要一组关于数据通路的操作。数据通路从存储器中获取一条指令，通过生成必要的控制信号来对指令进行译码操作，通过执行对数据通路的操作来执行指令。根据这些控制信号，对数据通路的操作可能还需要从存储器中获取数据，并将计算结果或来自存储器的数据（如果有）存储（写回）到寄存器中。寄存器中的内容可能存储到存储器中。

一般来说，每个不同的 CPU 都有一套独特的指令集。然而，英特尔和 AMD 的一些处理器采用了相同的指令集。例如，32 位的指令集系统 X86 可在英特尔和 AMD 的处理器上执行。其他知名的指令集例子有英特尔的 IA64（安腾架构）、AMD 的 X64（64 位指令集）、MIPS、Sparc 和 ARM。虽然每个指令集不同，但是每个指令集中的指令均可用于开发任何类型的软件，包括系统软件、单机和线上的应用软件。

伴随着晶体管数量的增加，现代 CPU 也通过流水线技术和指令级并行 (ILP) 技术来提升系统性能，正如我们在第 1 章中初步探讨过的。在这种情况下，流水线的数据通路也被称为**指令流水线**。当同时执行多条指令时，流水线的各级将一直保持忙碌状态，从而提升数据通路的效率。通过采用指令流水线技术增加了指令吞吐量和每秒执行的指令数，并减少了程序的总执行时间。

307

然而，当数据依赖的指令通过流水线时，如果必要，将需要额外的硬件来运行流水线，从而确保不违背数据之间的依赖关系。这也将导致流水线的效率降低，除非使用特定的硬件并进行编译优化来消除或降低这种数据依赖性。

此外，分支指令也会改变执行流程，引入气泡也会降低流水线效率。然而，现代 CPU 通过采用分支预测机制来最小化分支指令带来的影响。

随着指令级并行技术和指令流水线技术的使用，CPU 可在多个并行化的流水线中同时执行多条指令。哪些指令可同时执行取决于程序，并且由编译器（即在软件中）**静态**或在硬件中**动态**决定。指令级并行技术也能降低程序执行的总时间。然而，如在第 1 章中阐述的那样，程序在每个流水线周期内，只能执行有限数量的独立指令。此外，因为指令和数据必须来自存储器，许多现代的 CPU（例如，英特尔酷睿 i7）实现多线程，使它们在 CPU 中等待接收指令或数据时，可以切换到另一个执行程序（线程）。虽然这并不会减少程序总的执行时间，但这有助于提高处理器的整体效率，使处理器可执行更多任务并减少处理器的空闲时间。

在本章中，我们先介绍一些背景知识，并说明不同的指令集体系结构。然后，为了更好地理解指令集和数据通路的设计，我们将讨论一个简单的高级语言的代码示例。在这个示例中，将展示指令集和单周期数据通路。我们将提供一个使用硬件描述语言（HDL）来描述的单周期数据通路。我们将展示采用单周期数据通路和流水线数据通路对示例程序执行的模拟执行结果，并讨论其性能参数。

本章最后将介绍精简指令集计算机（RISC）结构及其优势，并介绍采用精简指令系统进行编译器优化和提高指令吞吐量的相关技术。特别是我们将通过示例来阐述增加流水线时钟频率、分支预测技术、指令级并行、多线程等相关技术，一个关于多线程的程序示例也同时提供给大家。

8.1.1 指令类型

处理器通常按通用编程要求进行设计。然而，为了实现更好的性能和支持一些实时应用，计算机系统经常需要使用专用处理器，例如图形处理器（GPU）和数字信号处理器（DSP）。各个专用处理器都有一组为高效地执行计算机图形（例如对象的旋转）、信号处理（例如音频压缩）等设计的指令集。DSP 通常用于嵌入式系统，如手机和数码相机，执行信号和图像处理等任务。现代处理器还可以包括某些特殊用途的指令，如单指令多数据（SIMD）（第1章）指令集，以及英特尔、AMD 处理器中与计算机安全相关的指令。

执行算术运算和逻辑运算的指令通常被称为**数据操作指令**，用于对数据进行计算操作。其他指令被称为**程序流控制指令**，如有条件和无条件分支（或跳转）指令和**数据移动指令**，以及那些用于读出和写入存储器的指令。程序流控制指令改变程序的执行路径，在高级语言语句的执行过程中，这些指令都是必要的，如“if-else”“for 循环”“while 循环”和子程序过程调用等。后者需要保存处理器的返回地址和状态（即寄存器的内容），可以保存在处理器内的一组特殊寄存器中（例如 Sparc 的寄存器窗口），也可以保存在存储器中（例如存储器栈）。（预知更多关于寄存器窗口的相关内容，请参考第9章的练习9.14。）

8.1.2 程序翻译

如图8-1所示，使用高级语言（如C/C++或Java）编写的软件程序被编译器翻译为汇编指令（C/C++）或字节码（Java），字节码在程序运行时转换为指令。汇编指令是由其助记符定义的。助记符是一种易于记忆的操作码（op-code），如加法操作使用ADD，减法操作使用SUB。有关助记符和汇编语言规则的详细信息，请参阅IEEE标准中的微处理器汇编语言[1]。

汇编器将给每一个助记符操作码分配一个独特的二进制数，汇编器也被用来将指令翻译为二进制数，生成一个**目标代码**，目标代码存储在磁盘上的一个文件中。两个或多个（如果有）目标文件将被链接起来生成一个可执行文件（如Windows环境中的.exe文件）。当静态库函数在程序中被调用时，链接程序将链接静态库函数（如math）的目标代码。此外，一个程序也会包含一些动态链接代码（即动态链接库），动态链接在程序执行时进行（未在图8-1中展示）。最后，操作系统程序装载机将加载可执行代码到存储器中执行程序。

8.1.3 指令周期

图8-2说明了一个指令执行的数据通路，这一数据通路也被称为**指令周期**，指令周期分

为取指令、译码、执行和写回 4 部分。数据通路可以以单周期、多周期或流水线的方式执行程序。从指令存储器 (IM) 中取出指令, 从数据存储器 (DM) 中读出或写入数据。由于用于操作现代处理器的时钟频率比同步动态随机读写存储器 (SDRAM) 的时钟频率高, 所以采用了静态随机读写存储器 (SRAM) 技术的高速缓冲存储器 (cache) 将作为 IM 和 DM 的主要部件。程序执行过程中, 先从 SDRAM 中复制指令和数据, 并将指令和数据传送到高速缓冲存储器中。高速缓冲存储器通过保存经常执行的指令 (例如循环操作) 和处理器经常访问的数据, 提升了系统的整体性能。关于高速缓冲存储器的相关内容将在第 10 章讲述。

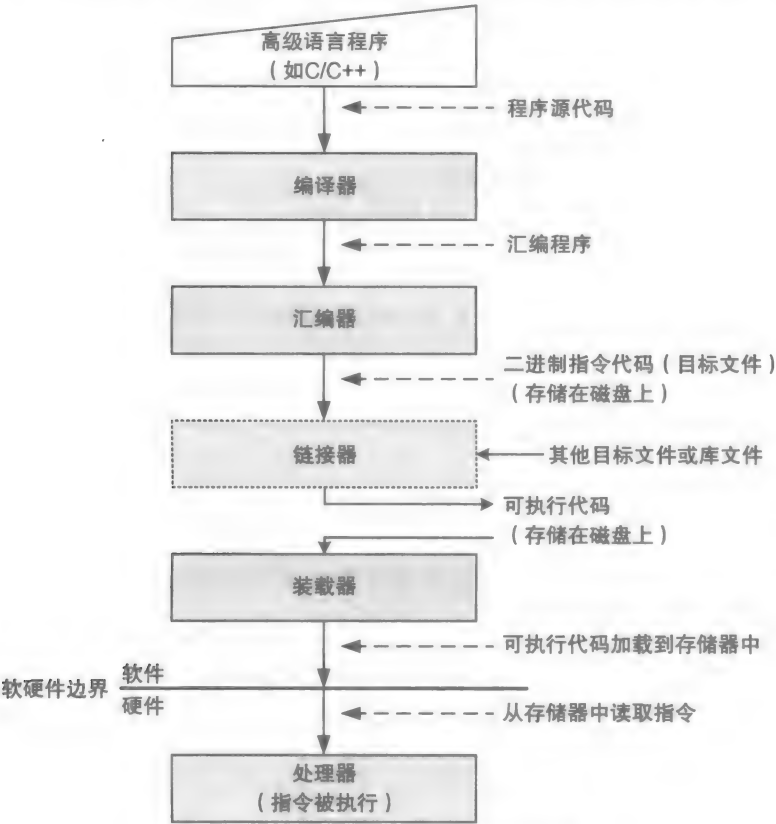


图 8-1 基本程序翻译和执行过程

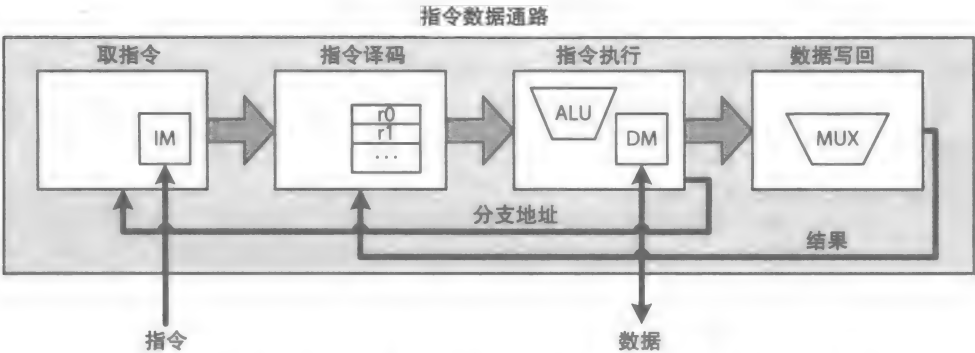


图 8-2 包含指令存储器 (IM) 和数据存储器 (DM) 的指令数据通路

8.2 指令集体系结构的类型

多年来，指令集体系结构的发展主要依据集成芯片（IC）技术和计算机结构，如流水线。除了操作码以外，指令还包括一些明确指定的或者隐含的或两者都有的操作数。操作数分为输入操作数和输出操作数，输出操作数一般为一个。输入操作数可指定为一个常量或一个寄存器或存储器的内容，这个常量也被称为立即数，如数字 9。输出操作数可以为一个寄存器号或者一个存储器地址。寻址模式用来解释指令执行过程中目标数据的获取方式，不同的寻址模式决定了不同的获取方式。

8.2.1 寻址模式

表 8-1 列出了多种寻址方式的示例。在这个表中，括号用于将立即值和存储器地址区分开。这些符号通过汇编指令转换为对应的二进制文件，这些符号通常被称为机器指令。一个立即（I）数是一个二进制的补码，其可在指令执行过程中直接使用。直接寻址（D）的操作数是一个存储器地址，其数据必须来自于存储器。变址（X）操作数确定了存储器中下一个数组元素的地址。此外还有一些其他寻址模式的例子，它们所对应的数据的存放地址在其他地方。

表 8-1 寻址模式的样例

| 操作数符号 | 寻址模式 |
|--------|---|
| V | I，立即数：V 是一个立即数形式的输入操作数，二进制补码 |
| (V) | D，直接寻址：V 是一个存储器地址，(V) 表示存储器地址为 V 的存储单元中的数据（即 $M[V]$ ） |
| R | R，寄存器：表明一个输入数据寄存器源或一个目的寄存器，或两者 |
| R, (V) | X，变址寻址：V 是一个寄存器地址， $R + V$ 是存储器中下一个数据的地址（即 $M[R + V]$ ） |

表 8-2 说明了使用明确指定的或隐含的操作数的多个指令例子。然而，这些指令并不全属于一个单独的处理器。每条指令都用于计算两个数据值的总和，并将总和存储在一个寄存器中。表中的第一条指令没有明确说明操作数，在这种情况下，两个数据的源地址和计算结果的目的地址存放在数据通路的堆栈上。第二条指令明确说明了一个操作数，为立即数 9，在这种情况下，指令隐含了一个寄存器，这个寄存器既被用来存储数据源地址，也被用来存储计算结果。

第三条指令包含两个明确指定的操作数，寄存器 R1 和立即数 - 9，在这种情况下，R1 不仅保存输入操作数，也保存输出的结果和，也就是说，指令执行 $R1 \leftarrow R1 + -9$ 。在第 4 个例子中，第二个操作数是一个存储器地址，指令执行 $R1 \leftarrow R1 + M[9]$ 操作，其中 $M[9]$ 表示存储器地址为 9 的存储单元中存储的数据。在第 5 个例子中，第二个和第三个操作数表明了第二个输入数据的存储地址，为 $R2 + 9$ ，指令执行 $R1 \leftarrow R1 + M[R2 + 9]$ 。在第 6 个例子中，两个输入操作数为寄存器中存储的数据，指令执行 $R1 \leftarrow R1 + R2$ 。在第 7 个例子中，明确指明了目的寄存器，目的寄存器的寄存器号可以和两个源地址寄存器号之一相同，也可以不同，指令执行 $R3 \leftarrow R1 + R2$ 。

[311]

表 8-2 一系列包含明确指定的或隐含的操作数的指令示例

| 示 例 | 指 令 | 描 述 |
|-----|-------|--|
| 1 | ADD | 源操作数和目的操作数已被指令所知（如：硬件栈） |
| 2 | ADD 9 | 立即数寻址： $R \leftarrow R + 9$ ；R 是一个指令已知的寄存器 |

(续)

| 示 例 | 指 令 | 描 述 |
|-----|-----------------|---|
| 3 | ADD R1, -9 | 寄存器和立即数: $R1 \leftarrow R1 + -9$; R1 既被用来存储源寄存器号, 也被用来存储目的寄存器号 |
| 4 | ADD R1, (9) | 寄存器和直接寻址: $R1 \leftarrow R1 + M[9]$ |
| 5 | ADD R1, R2, (9) | 寄存器和变址寻址: $R1 \leftarrow R1 + M[R2 + 9]$ |
| 6 | ADD R1, R2 | 寄存器和寄存器 (两个操作数): $R1 \leftarrow R1 + R2$ |
| 7 | ADD R3, R1, R2 | 寄存器和寄存器 (三个操作数): $R3 \leftarrow R1 + R3$; 目的寄存器可以和两个源寄存器不同 |

8.2.2 指令格式

指令格式用来把助记的汇编指令转换到机器指令。指令格式指定了一个操作码、寻址模式、源寄存器 (如果有)、目的寄存器 (如果有)、 n 位的立即数 (如果有) 以及存储器地址 (如果有) 所需的位数。图 8-3 说明了表 8-2 中列出的指令的指令格式。

例如, 表 8-2 中的包含 8 位的操作码、16 位的寄存器以及 16 位的立即数的指令 3 (ADD, r1, -9), 在假定 ADD 操作码为 $(00000001)_2$ 、寄存器和立即数 (RI) 寻址模式为 $(1000)_2$ 的情况下, 指令译码为 4B 机器指令的格式如下:

```
0000 0001, 1000, 0001, 1111 1111 1111 0111 or
0181FFF7 in hex
```

不同指令格式的大小和数量取决于 ISA 的类型。总体而言, ISA 分为堆栈 ISA、累加器 ISA (Acc-ISA)、CISC-ISA 和 RISC-ISA, 这将在下文讨论。

312

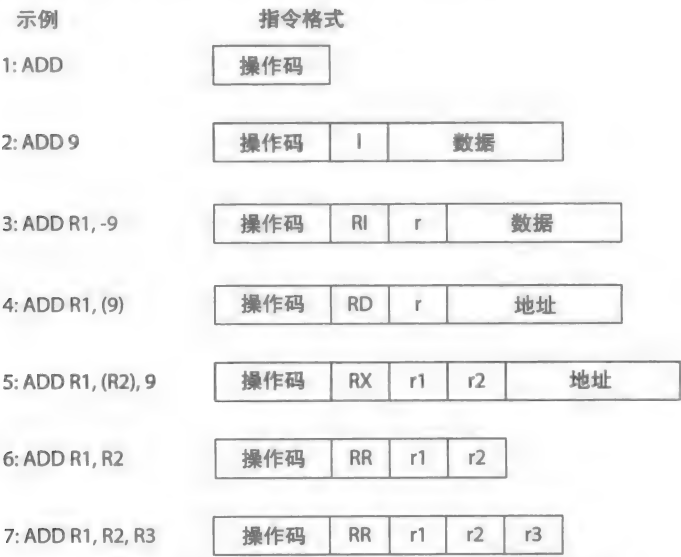


图 8-3 表 8-2 中列出的 8 条指令的指令格式示例

8.2.3 堆栈 ISA

采用堆栈 ISA 的处理器使用一个硬件堆栈来操作数据以后进先出 (LIFO) 的顺序执

行，即最后一个存储的数据将是第一个取回的数据。数据从存储器中取出，压入堆栈中，用于计算的数据或者需要重新写回到存储器的数据需从堆栈中弹出，计算的结果也压入堆栈中去。

使用堆栈 ISA 的优势在于其指令长度在绝大多数指令中最短。因此，当处理器输入输出管脚有限时，堆栈 ISA 是处理器设计的理想架构。该处理器将像惠普（HP）计算器一样运行，需要将一个数学公式首先按逆波兰式进行转换。例如，在惠普计算器中计算 $2 \times (3 + 4)$ ，必须先输入 3，然后是 4，再是 + 号，然后是 2，最后是 *。思考下面的高级语言程序语句：

```
A = B * (C + D);
```

为了让该语句可在堆栈 ISA 处理器上执行，编译器必须先将语句转换为逆波兰式 $CD + B* = A$ 。接着编译器把符号转换到下面的堆栈 ISA 示例汇编程序。注意，算术指令 ADD 和 MULT 不需要操作数，这使得每条指令的长度非常短。压入指令（PUSH）、弹出指令（POP）以及数据移动指令（MOV）都只需要一个操作数。总体而言，程序的控制顺序指令，比如分支指令或子程序调用，也只需要一个操作数。

[313]

```
Instruction
number
1:    PUSH (C) //stack←M[C]
2:    PUSH (D) //stack←M[D]
3:    ADD      //stack←(C) + (D), values popped, added,
              //result pushed
4:    PUSH (B) //stack←M[B]
5:    MUL      //stack←((C) + (D)) * (B), values popped, added,
              //result pushed
6:    POP (A)  //M[A]←((C) + (D)) * (B)), value is popped
              //and stored in memory
```

图 8-4 阐明了上述程序使用堆栈 ISA 处理器的执行过程。在示意图中，假定 $(B) = 2$ ， $(C) = 3$ ， $(D) = 4$ ，展示了指令执行过程中逻辑堆栈和硬件堆栈的内容。

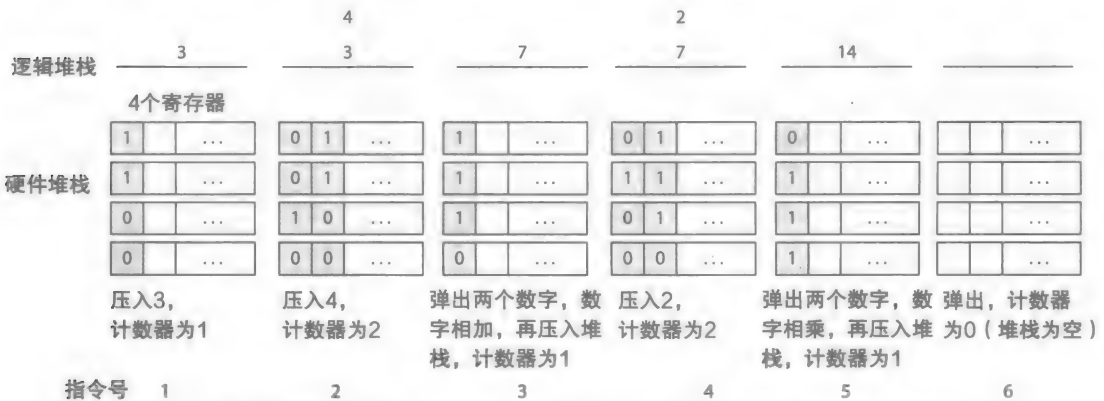


图 8-4 计算逆波兰式 $CD + B* = A$ 时的堆栈内容，假定 $(B) = 2$ ， $(C) = 3$ ， $(D) = 4$

堆栈 ISA 的劣势在于在之后的计算过程中不能重复使用存储器的内容。一旦存储器内容从堆栈中弹出，在处理器中将不再可用。例如，考虑程序语句 “ $A = (B + C) * (B + D);$ ”，要求变量 B 两次压入堆栈中，这增加了存储器的通信量。

8.2.4 累加器 ISA

累加器 ISA 是最简单的架构，并且需要的硬件最少。数据通路包含一个专用的既作为源寄存器也作为目的寄存器的寄存器，这个寄存器叫累加器（ACC）。寄存器被用来作为一个隐含输入操作数和一个隐含输出寄存器名。例如，考虑高级语言程序语句“ $A = B * (C + D);$ ”，其采用累加器 ISA 的示例汇编程序如下所示。相对于堆栈 ISA，在累加器 ISA 中，算术指令可直接操控存储器中的数据，因此，相对于堆栈 ISA，累加器 ISA 减少了转换高级语言程序为汇编程序的指令数量。 [314]

```
1:  LD      (C)      //ACC ← M[C]
2:  ADD     (D)      //ACC ← ACC + M[D]
3:  MUL     (B)      //ACC ← ACC * M[B]
4:  ST      (A)      //M[A] ← ACC
```

然而，累加器 ISA 的缺点也正是因为 ACC 而成为架构的瓶颈，就像堆栈 ISA 中的硬件堆栈一样。ACC 中的内容可能需要重新写入存储器中以便下一个计算使用。例如，考虑计算 $A = (C + D) * (E - F)$ ，其中 $C + D$ 操作和 $E - F$ 操作必须在乘操作之前得到结果。在这种情况下，一旦 $C + D$ 进行了计算，在 ACC 中保留的和必须写回到一个临时的存储器单元，以便可以计算 $E - F$ 。累加器 ISA 也可以包含其他的寄存器，比如访问存储器中存储单元的变址寄存器以及保存子程序调用的返回地址的链接寄存器。

8.2.5 CISC-ISA

复杂指令集计算机（CISC）ISA 是在累加器 ISA 的基础上，通过增加数据通路中工作寄存器的数量得来的。每个算术指令可包括一个或两个显式声明的寄存器操作数。和累加器 ISA 相似，CISC-ISA 的算术指令可直接从存储器中引用数据。此外，CISC-ISA 一般实现多种寻址模式，需要不同尺寸的多种指令格式。

算术指令的输入操作数可以为一个立即数、寄存器或者存储器内容，不过操作数中只可以有有一个是立即数或存储器内容。通常，一个 CISC-ISA 处理器可实现很多简单和复杂的指令，因此，将一个高级语言转换为 CISC 指令所需要的指令数量最少。然而，一些复杂指令可能需要更多时间来执行。下面是计算语句“ $A = B * (C + D);$ ”采用 CISC-ISA 的汇编程序示例：

```
1.  LD      R1,      (C)      //R1 ← M[C]
2.  ADD     R1,      (D)      //R1 ← R1 + M[D]
3.  MUL     R1,      (B)      //R1 ← R1 * M[B]
4.  ST      (A),     R1      //M[A] ← R1
```

因为有更多的寄存器可以选择，语句“ $A = (C + D) * (E - F);$ ”可使用一个寄存器（如 R1）来计算 $C + D$ ，用另一个寄存器（如 R2）来计算 $E - F$ ，然后将两个寄存器的内容相乘生成最终结果。然而，即使在 CISC-ISA 中有多个工作寄存器，相比于一个典型的高级语言程序中的变量数量，可用的寄存器数量还是非常少。并且，取决于每个高级程序语句的尺寸，可能有许多中间结果（例如， $C + D$ 和 $E - F$ ）要保存在处理器内部的寄存器或存储器中。因此，与累加器 ISA 相似，在寄存器中的中间结果可能也需要写回到存储器中从而释放寄存器以方便下一条计算使用。 [315]

[与堆栈 ISA 或累加器 ISA 编译器不同的是，当所有的寄存器包含中间结果时，CISC-

ISA 编译器需要实现一个寄存器选择策略，以尽量减少内存访问。寄存器分配策略，如最近最少使用（LRU），用来将一个存储了中间结果的寄存器的内容写回存储器，从而释放该寄存器。另一方面，如果 LRU 寄存器中的内容是存储器数据，则直接丢弃其中内容。]

8.2.6 RISC-ISA

如前所述，一个典型的 CISC-ISA 数据通路使用多种寻址模式。另一方面，RISC-ISA 的设计者认为更简单的指令可使得数据通路更简单、更有效。阿尔法、MIPS 和 Sparc 的处理器从一开始就是依照 RISC-ISA 处理器进行设计的。

RISC-ISA，也被称为**加载/存储结构**，只使用两条指令（加载指令 LD 和存储指令 ST）来访问存储器。算术指令不能直接在存储器上操作数据。存储器数据必须在其进行计算之前加载到寄存器中。下面是计算语句“ $A = B * (C + D)$ ；”采用 RISC-ISA 的汇编程序示例：

```

1.  LD   R1,   (C)       //R1 ← M[C]
2.  LD   R2,   (D)       //R2 ← M[D]
3.  ADD  R3,   R1, R2     //R3 ← R1 + R2
4.  LD   R4,   (B)       //R4 ← M[B]
5.  MUL  R5,   R3, R4     //R5 ← R3 * R4
6.  ST   (A),  R5        //M[A] ← R5

```

这个汇编程序在处理器中使用 5 个寄存器来保存 3 个存储器内容和两个中间变量（即 $C * D$ 和 $B * (C + D)$ ）。相对于 CISC-ISA 处理器来说，通常 RISC-ISA 处理器使用更多的寄存器，这样是因为要在寄存器中保存更多的数据，以便提高处理器的吞吐量。

8.3 设计示例

上文已给出一个简单的累加器 ISA 的 CPU 设计示例，然而，这里的目的并不是要建立一个完整的指令集，而是要提供一种自顶向下的设计方法。我们从一个简单的高级语言示例来阐述这种方法，其需要包含以下内容：

- 指令集设计（一个示例高级语言程序代码转换为其等价的汇编程序所必需的指令清单）。
- 汇编语言程序代码清单。
- 可执行二进制代码（机器指令）。
- 数据通路设计。
- 硬件描述语言（HDL）模型。
- 仿真。

316

8.3.1 累加器 ISA 指令集设计

例 8-1 展示了使用 for 循环求解数组中各元素之和的高级语言程序代码。我们将建立一个累加器 ISA 指令集来编译代码，从而生成等价的汇编程序。

例 8-1 求解数组大小为 8 的元素之和的程序代码：

```

int array[8];
int i, sum;

```

```
sum = 0;
for (i = 0; i < 8; i++)
    sum = sum + array[i];
```



仔细检查示例代码可知，我们需要创建算术指令（如加法和比较操作）、包括数组索引在内的数据移动指令以及程序流控制指令（如跳转、比较结果为大时跳转等）。对于数组索引，数据通路必须包含另一个寄存器（X）来保存数据元素的下一个索引。表 8-3 展示了翻译示例程序代码为等价的汇编语言程序所需的指令集。

317

在这个表中，**程序指针（PP）**也叫作**程序计数器（PC）**，其保存程序执行的下一指令所在的地址。这里，整数从 0 开始任意分配给操作码。操作码 0 被命名为空操作（NOP）。虽然这里使用空操作，但它是设计高性能 RISC 处理器的必要部分。

表 8-3 转换高级语言程序为等价的汇编语言程序的累加器 ISA 指令集示例

| 操作码 | 指 令 | 寻址方式 | 示 例 | 动 作 |
|-----|-----|-------|----------------|--|
| 0 | NOP | | NOP | 不做任何事情 |
| 1 | ADD | 立即寻址 | ADD data | ACC ← ACC + data |
| 2 | | 直接寻址 | ADD (address) | ACC ← ACC + M[address] |
| 3 | CMP | 立即寻址 | CMP data | if ACC == data then GTF = 1 else GTF = 0 |
| 4 | JGT | 立即寻址 | JGT address | PP ← address if GTF = 1 |
| 5 | JMP | 立即寻址 | JMP address | PP ← address |
| 6 | LD | 立即寻址 | LD data | ACC ← data |
| 7 | | 直接寻址 | LD (address) | ACC ← M[address] |
| 8 | | 变址寻址 | LD X (address) | ACC ← M[X + address] |
| 9 | MVX | 寄存器寻址 | MVX | X ← ACC |
| 10 | ST | 直接寻址 | ST (address) | M[address] ← ACC |

ACC：累加器；GTF：大于标志位；PP：程序指针；X：变址寄存器

1. 累加器 ISA 汇编程序示例

由于没有针对累加器 ISA 处理器的编译器，在例 8-1 中的代码必须手动转换为汇编代码，如例 8-2 所示。累加器 ISA 汇编语言程序的每一行中包含 4 个字段：一个可选的标签字段，一个操作码字段，一个操作数字段（如果有），一个可选的注释字段。标签字段可能包含一个跳转地址，如例 8-2 中的 L1 和 L2。

“**.code**”和“**.data**”是用于分离程序中指令和数据的汇编指令。当处理器数据通路要求在执行过程中指令和数据要存放在不同的存储器区域中时，这两个命令是必要的。为了实现更高的性能，指令和数据将分别存储在处理器的 IM 和 DM 中，如图 8-2 所示。示例中代表保留字节的“RB”，是一个伪指令的示例。伪指令“RB”被编译器解释为给程序中的变量 sum 和 i 以及数据结构 array 分配存储器地址空间。

例 8-2 例 8-1 中的累加器 ISA 汇编语言程序：

```
.code //start program code
LD 0 //Initialize, ACC ← 0
ST (sum) //M[sum] ← ACC
ST (i) //M[i] ← ACC

L1:
CMP 7 //is i > 7? (is ACC == 7?)
JGT L2 //exit for-loop if yes (PP ← L2)
```

318

```

MVX          //get next index (X←ACC)
LD   X(array) //get next array element (ACC←M[array
              //+ X])
ADD  (sum)    //and add it to the partial sum (ACC←ACC
              //+M[sum])
ST   (sum)    //store the partial sum in memory (M[sum]
              //← ACC)
LD   (i)      //do i = i + 1: get i (ACC←M[i]),
ADD  1        //increment i (ACC←ACC + 1), and
ST   (i)      //save i (M[i] ←ACC).
JMP  L1       //loop back

L2: ...

.data //start program data
array: RB 8 //reserve 8 bytes for array in memory
i:     RB 1 //reserve 1 byte for i in memory
sum:   RB 1 //reserve 1 byte for sum in memory
```

例 8-2 中的程序使用的语法是自己定义的。然而，对于一个特定语法的例子，请参考微软汇编器 (MASM) [2]。

2. 代码和数据存储空间

图 8-5 展示了程序执行过程中代码和数据是如何存储在虚拟存储器中的。虚拟地址的范围决定了程序的最大大小，单位为字节。例如，对于一个 32 位的用户虚拟地址空间，用户程序最大可以为 4GB。

分配给一个正在运行的程序的虚拟地址空间被划分为程序代码区和程序数据区，如图 8-5 所示。虚拟存储系统将在第 10 章讲述。然而，这里我们假定了图 8-5 表示了示例程序在虚拟存储空间分配的方式。

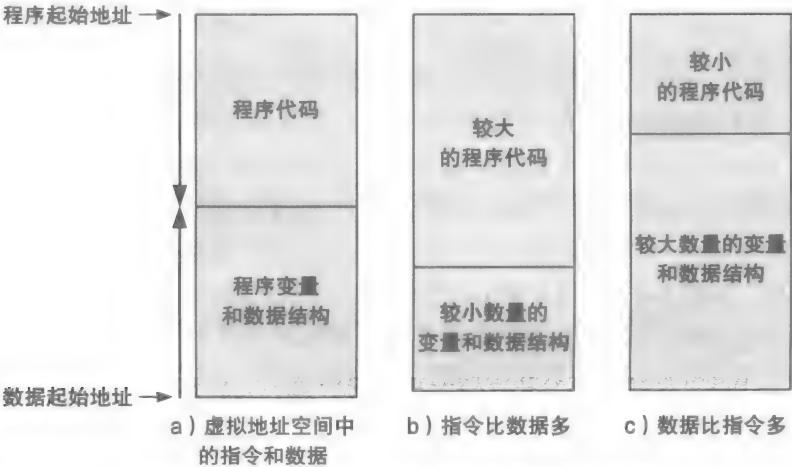


图 8-5 虚拟存储地址空间中的程序代码和数据

3. 基于奔腾 IV 指令架构的汇编程序示例

例 8-3 展示了对例 8-1 中的程序采用英特尔奔腾 IV 指令架构翻译得到的汇编代码，这是使用 CygWin 中的 gcc 编译器编译生成的，CygWin 是在 Windows 环境下与 Linux 环境下相同效用的编译器 [3]。在列表中，整数数据类型大小为 4B。在每条指令最右侧的操作数表示目的操作数。在程序中，寄存器具有前缀标记“%”。寄存器 %ebp 被称为基指针，其维持一个程序数据区在存储器中的起始地址。并且，为了避免 cache 和存储器未对齐以及实现更好

319

的性能 [4], 将先分配大容量数据结构。然而, 随着更大容量的主存储器的使用, 数据结构可通过声明比必要 (填充的) 容量更大的空间来达到更好的高速缓冲性能 [5]。在这个代码中, 数据地址 `%ebp - 40` 被赋值给 `array`, `%ebp - 44` 被赋值给变量 `i`, `%ebp - 48` 被赋值给 `sum`。读入有效地址 (“`leal`”) 指令读入一个存储器地址 (不是存储器内容) 到一个寄存器中。

例 8-3 例 8-1 中的示例程序的基于奔腾 IV 指令架构的汇编代码:

```

movl $0, -48(%ebp)    //initialize sum, Memory[%ebp - 48] ← 0 (sum = 0)
movl $0, -44(%ebp)    //initialize i, Memory[%ebp - 44] ← 0 (I = 0)
L7:
    cmpl $7, -44(%ebp) //is i > 7?
    jg  L8             //if yes, jump to L8
    movl -44(%ebp), %eax //get i, %eax ← Memory[i]
    movl -40(%ebp,%eax,4), %edx //get array[i], %edx ← Memory[array + i + 4]
    leal -48(%ebp), %eax //get address of sum, %eax ← %ebp - 48
    addl %edx, (%eax)    //Memory[sum] ← array[i] + Memory[sum]
    leal -44(%ebp), %eax //get address of i, %eax ← %ebp - 44
    incl (%eax)         //Memory[i] ← Memory[i] + 1
    jmp L7              //repeat
L8: ...

```

4. 基于 Sparc 架构下的示例汇编程序

例 8-4 展示说明了例 8-1 中的程序经过 gcc 编译器编译生成的基于 AltraSparc II 指令架构的汇编语言程序。在这个示例中, gcc 编译器是在 Virtutech Simics 环境下的虚拟 Aurara SPARC Linux 操作系统中使用的 [6]。

AltraSparc 是一个 RISC-ISA 的处理器, 因此, 算术指令不会直接访问存储器, 它只操纵寄存器中的内容和立即数。只有读入 (`ld`) 或存储 (`st`) 指令可访问存储器。寄存器采用前缀标记 “%”, 寄存器 “`%fp`” 表示帧指针, 和英特尔架构中的基指针相似, 保存存储器数据区的基地址。因为 Sparc 处理器自动执行紧随分支指令后下一指令, 比如 “`bg`” 和 “`b`”, 未被优化的代码必须在每个分支指令后包含一个空操作指令。

正如预期的那样, 例 8-4 中的 RISC 程序的指令数量 (19 条) 要比例 8-3 中的 CISC 程序中指令数量 (11 条) 多。

320

例 8-4 例 8-1 中示例程序在 Sparc 架构下的汇编代码, 代码未被优化。

```

st    %g0, [%fp-48]    //Store, Memory[fp - 48] ← g0 (g0 always 0) (sum = 0)
st    %g0, [%fp-44]    //Store, Memory[fp - 44] ← g0 (i = 0)
.LL5:
    ld    [%fp-44], %g1 //Load, g1 ← Memory[i]
    cmp   %g1, 7        //Compare, is g1 > 7?
    bg    .LL6          //Branch if greater than 7
    nop
    ld    [%fp-44], %g1 //Load, g1 ← Memory[i]
    sll   %g1, 2, %g2    //Compute ptr to array[i]: Shift Left Logical (i * 2)
    add   %fp, -8, %g1   //g1 ← fp - 8 (get memory location of array)
    add   %g2, %g1, %g1  //g1 ← g2 + g1 (array location + next i * 2)
    ld    [%fp-48], %g2 //Load sum, g2 ← Memory[fp - 48]
    ld    [%g1-32], %g1  //Load array[i], g1 ← Memory[g1 - 32]
    add   %g2, %g1, %g1  //Array[i] + sum, g1 ← g2 + g1
    st    %g1, [%fp-48]  //Store sum, Memory[fp - 48] ← g1
    ld    [%fp-44], %g1  //Load i, g1 ← Memory[fp - 44]
    add   %g1, 1, %g1    //Increment, g1 ← g1 + 1
    st    %g1, [%fp-44]  //Store i, Memory[fp - 44] ← g1
    b     .LL5          //Branch to instruction ay LL5
    nop
.LL6:

```

5. 可执行代码

一个汇编语言程序将被汇编器翻译成等价的二进制代码，一般需要进行两次扫描。在第一次扫描过程中，汇编器给代码和数据区中的每个标记分配一个存储器地址。表 8-3 中所列的指令使用图 8-6 中所示的三种指令格式来表示其二进制指令。为了简单起见，每条指令由 8 位的操作码和 8 位的数据或 8 位的地址操作数（如果有）构成。指令“NOP”和“MVX”没有操作数，因此，这两条指令中的操作数字段设置为 0。

例 8-5 列出了例 8-2 中程序的典型的汇编器输出结果，列表中使用自定的语法。为了简单起见，示例累加器 ISA 处理器假定为 16 位的机器，并假定程序代码和数据的地址空间为 256B，程序代码起始地址为 0，数据起始地址为 0xFF（一个基地址）。

[321]



图 8-6 示例累加器 ISA 处理器的指令格式

例 8-5 对例 8-2 中汇编程序手动解释输出结果。array 起始地址 0xF0，大小为 16B，变量 i 起始地址 0xEE，大小为 2B，变量 sum 起始地址 0xEC，大小为 2B。

| Address | Instruction | Instruction in binary | Inhex |
|---------|-------------|-----------------------|-------|
| 0: | LD 0 | 0000,0110;0000,0000 | 0600 |
| 2: | ST 0xEC | 0000,1010;1110,1100 | 0AEC |
| 4: | ST 0xEE | 0000,1010;1110,1110 | 0AEE |
| 6: | CMP 7 | 0000,0011;0000,0111 | 0307 |
| 8: | JGT 0x1A | 0000,0100;0001,1010 | 041A |
| A: | MVX | 0000,1001;0000,0000 | 0900 |
| C: | LD X(0xF0) | 0000,1000;1111,0000 | 08F0 |
| E: | ADD (0xEC) | 0000,0010;1110,1100 | 02EC |
| 10: | ST (0xEC) | 0000,1010;1110,1100 | 0AEC |
| 12: | LD (0xEE) | 0000,0111;1110,1110 | 07EE |
| 14: | ADD 1 | 0000,0001;0000,0001 | 0101 |
| 16: | ST (0xEE) | 0000,1010;1110,1110 | 0AEE |
| 18: | JMP 6 | 0000,0101;0000,0110 | 0506 |
| 1A: | ... | | |

■

例 8-6 对例 8-3 中汇编程序手动解释的输出结果。英特尔奔腾系列处理器执行不同大小的 CISC 指令。例如，“movl”是一条 7B 大的指令，“jg”是一条 2B 大的指令。

例 8-6 例 8-3 中奔腾 IV 程序的汇编器输出结果。数字 0xfffffd0、0xfffffd4 和 0xfffffd8 表示二进制补码形式的数字 -48、-44 和 -40。

| | | | |
|---------|----------------------|------|-----------------------------|
| 401340: | c7 45 d0 00 00 00 00 | movl | \$0x0,0xfffffd0(%ebp) |
| 401347: | c7 45 d4 00 00 00 00 | movl | \$0x0,0xfffffd4(%ebp) |
| 40134e: | 83 7d d4 07 | cmpl | \$0x7,0xfffffd4(%ebp) |
| 401352: | 7f 13 | jg | 401367 <_main+0x77> |
| 401354: | 8b 45 d4 | mov | 0xfffffd4(%ebp),%eax |
| 401357: | 8b 54 85 d8 | mov | 0xfffffd8(%ebp,%eax,4),%edx |
| 40135b: | 8d 45 d0 | lea | 0xfffffd0(%ebp),%eax |
| 40135e: | 01 10 | add | %edx, (%eax) |
| 401360: | 8d 45 d4 | lea | 0xfffffd4(%ebp),%eax |
| 401363: | ff 00 | incl | (%eax) |

```
401365: eb e7                jmp     40134e <_main+0x5e>
401367: b8 02 00 00 00      ...
```

例 8-7 是对例 8-4 中 AltraSparc II 程序使用汇编器解释的输出结果。因为 Sparc 是一个 RISC-ISA，为了高效率地执行指令，所有指令都是相同长度。

例 8-7 例 8-4 中 AltraSparc II 程序使用汇编器解释的输出结果：

```
...104f4: c0 27 bf d0      clr    [ %fp + -48 ]
104f8: c0 27 bf d4      clr    [ %fp + -44 ]
104fc: c2 07 bf d4      ld     [ %fp + -44 ], %g1
10500: 80 a0 60 07      cmp    %g1, 7
10504: 14 80 00 0f      bg     10540 <main+0x90>
10508: 01 00 00 00      nop
1050c: c2 07 bf d4      ld     [ %fp + -44 ], %g1
10510: 85 28 60 02      sll    %g1, 2, %g2
10514: 82 07 bf f8      add    %fp, -8, %g1
10518: 82 00 80 01      add    %g2, %g1, %g1
1051c: c4 07 bf d0      ld     [ %fp + -48 ], %g2
10520: c2 00 7f e0      ld     [ %g1 + -32 ], %g1
10524: 82 00 80 01      add    %g2, %g1, %g1
10528: c2 27 bf d0      st     %g1, [ %fp + -48 ]
1052c: c2 07 bf d4      ld     [ %fp + -44 ], %g1
10530: 82 00 60 01      inc    %g1
10534: c2 27 bf d4      st     %g1, [ %fp + -44 ]
10538: 10 bf ff f1      b      104fc <main+0x4c>
1053c: 01 00 00 00      nop
10540: ...
```

322

8.3.2 累加器 ISA 处理器：单周期

图 8-7 以累加器 ISA 处理器为例说明了单个指令周期的数据通路，包括取指令、指令译码、指令执行和写回数据 4 个单元。在每个时钟周期中，指令被获取、被译码、被执行，指令执行的结果（如果有）写回到寄存器 ACC、X 或 1 位状态寄存器（SR），假定指令已经读入 IM（指令高速缓冲存储器）并且数据已经读入 DM（数据高速缓冲存储器）中。

并且，在每个时钟周期中，寄存器 PP 在程序中要执行的下一指令为顺序执行的指令时，将加载加法器的输出结果，而当程序要执行的下一指令为跳转指令的结果，如“JGT”或“JMP”时，PP 将加载复用器的输出结果。在图中，假定了 PP 保留的是字节地址，而 IM 输入一个字地址并输出下一个 16 位的指令。

译码单元包含一个组合电路，电路输入一个操作码，根据操作码生成所有必要的控制信号来执行当前指令。译码单元也包含寄存器 ACC、X、SR。总体而言，SR 是一个多位寄存器，每一位表示一种情况，比如相等（E）、小于（L）、不相等（NE）、算术运算输出结果为 0（Z）、有进位（C）、算术运算结果溢出（O）以及其他各种与处理器状态相关的状态。译码单元输出所有的控制信号、单独的立即数操作数（如果有）以及寄存器中的内容给执行单元。

执行单元包含关于执行指令的所有必要模块。当一个模块中有多个数据源时，MUX 是必需的。例如，加法器（Add）模块计算输入之和或比较器（Cmp）比较其输入数据，都输出是否大于标志位（gtf）的信号。模块输入为 ACC 和一个立即数或直接寻址的操作数。因此，当操作数为立即数时，需选择操作数，当操作数是一个表示存储器内容存放在的地址时，需寻址来获得操作数，如何选择需要通过一个复用器（MUX）来实现。同样，当寻址为直接寻址和变址寻址时，也需要一个复用器来选择获取操作数的方式。（参考第 3 章练习部分，如何设计一个比较器。）

写回单元包括一个复用器，它是用于选择执行单元生成的结果。“LD”和“ADD”指令的执行结果将存储（写回）到 ACC 中；“MVC”指令的执行结果将存储到寄存器 X 中，执行“CMP”，其结果是 1 位的大于标志位 gtf，结果存储在 SR 中的某 1 位中。执行“JMP”指令将加载下一条要执行指令的指令地址到 PP 中。

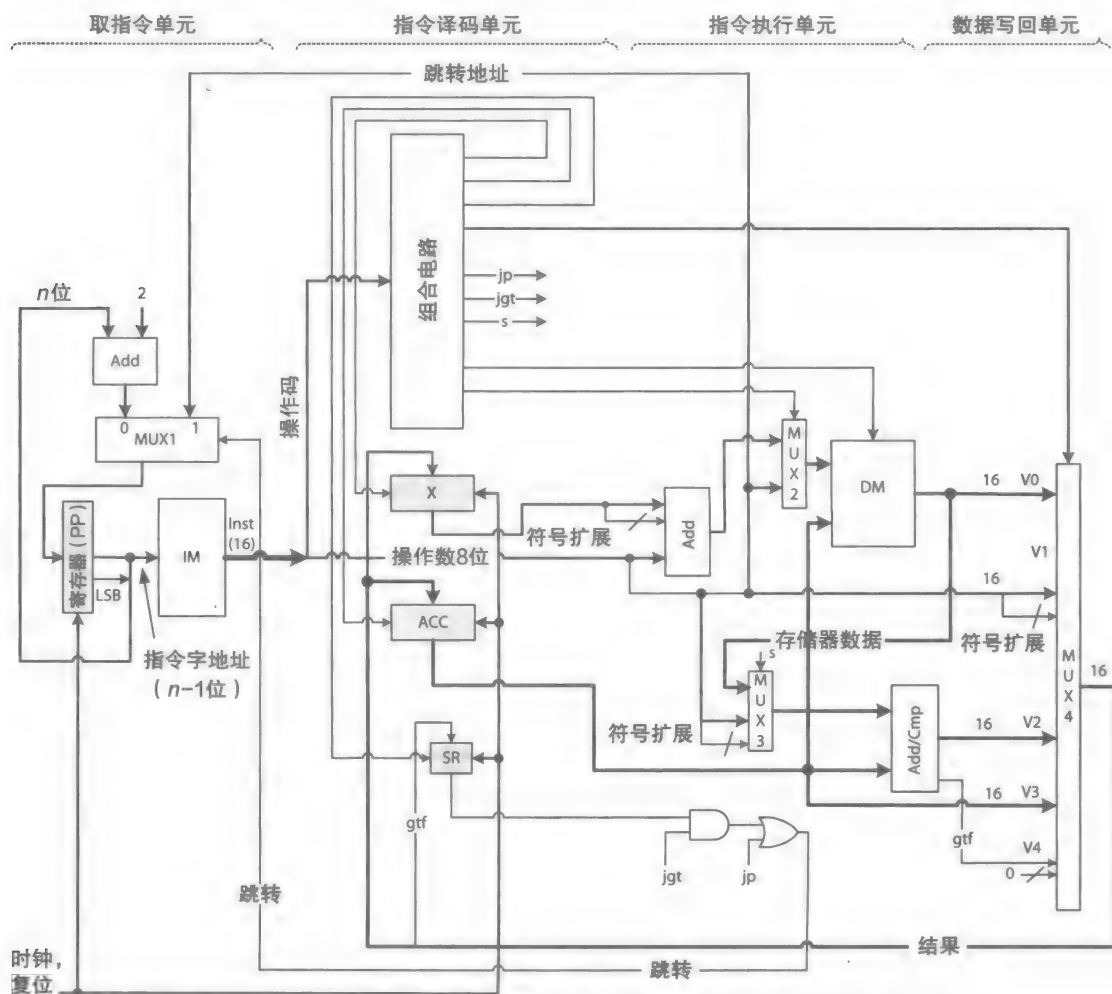


图 8-7 执行例 8-2 中的程序对应的累加器 ISA 单指令周期的数据通路

仿真

例 8-8 展示了图 8-7 中说明的单指令周期累加器 ISA 数据通路的 HDL 模型。为了简单起见，IM 和 DM 假定均为 64B，组织成 32×16 的结构。在一个实际的处理器中，IM 和 DM 都是由 cache 存储器构成（第 10 章）。由于处理器采用单周期数据通路，指令需要在—一个时钟周期内执行完。

例 8-8 对单周期累加器 ISA 数据通路的 HDL 行为描述包含两个初始化模块，初始化模块是用来初始化包含程序指令 IM 和包含数组数据的 DM。DM 使用 8 个元素，100 ~ 107 来进行初始化。一条额外的指令“JMP 0x1A”将插入程序的末尾，从而生成一个无限循环以达到仿真目的。包含两个主要代码区的描述将在接下来讨论。

译码和执行单元这种组合行为的描述隐含地说明了执行每条指令所需的控制信号，该代码还描述了创建数据通路所必需的组合电路。然而，HDL 模块并没有明确描述任何计算电路模型，其实现细节留给 Altera 的综合工具来决定。

取指令和写回单元这些单元通过改变寄存器 PP、ACC、X 和 SR 的内容来更新处理器的状态。因此，它们被组合起来描述，但是每个也可以分开描述。组合单元包含了加法器、用于取指令单元的复用器以及用于写回单元的复用器的行为描述，如图 8-7 所示。

```

module accISA(
    input clock, reset,
    output [15:0] inst,
    output reg [15:0] acc,
    output reg [4:0] x,
    output reg sr
);

    reg [5:0] pp; //program pointer, points to the next instruction
    reg [7:0] opcode;
    reg [15:0] operand;
    reg [15:0] result;
    assign inst = {opcode, operand[7:0]};
    wire [4:0] imAddress = pp[5:1]; //IM is 32 X 16, addressing one instruction word
    wire [5:0] dmAddress = operand[5:1]; //DM is 32 X16, addressing one data word

    (* ramstyle = "M512" *) reg [15:0] IM[0:31]; //instruction memory
    (* ramstyle = "M512" *) reg [15:0] DM[0:31]; //data memory

    parameter
        NOP = 0,
        ADD_I = 1,
        ADD_D = 2,
        CMP_I = 3,
        JGT = 4,
        JMP = 5,
        LD_I = 6,
        LD_D = 7,
        LD_X = 8,
        MVX = 9,
        ST = 10;

    //Initialize IM with machine instructions
    initial begin //Note, memory is organized as 31 X 16
        IM[0] = 16'h0600; //LD 0
        IM[1] = 16'h0AEC; //ST (0xEC) // (sum)
        IM[2] = 16'h0AEE; //ST (0xEE) // (i)
        IM[3] = 16'h0307; //CMP 7
        IM[4] = 16'h041A; //JGT 0x1A
        IM[5] = 16'h0900; //MVX
        IM[6] = 16'h08F0; //LD X(0xF0) //(array)
        IM[7] = 16'h02EC; //ADD (0xEC) //(sum)
        IM[8] = 16'h0AEC; //ST (0xEC) //(sum)
        IM[9] = 16'h07EE; //LD (0xEE) //(i)
        IM[10] = 16'h0101; //ADD 1
        IM[11] = 16'h0AEE; //ST 0xEE //(i)
        IM[12] = 16'h0506; //JMP 6
        IM[13] = 16'h051A; //JMP 0x1A, extra instruction to cause looping end
    //Initialize the array
    initial begin //Note, memory is organized 32 X 16
    //Therefore, for example, address 0xF0 = (1111 0000) is reduced to 0x30 = (11
    0000)
    //and then to 0x18 = (1 1000), dropping the LSB to access a 16-bit word
    //DM[8'h16] reserve for "sum"
    //DM[8'h17] reserved for "i"

```

```

DM[8'h18] = 100;          //array[0] = 100
DM[8'h19] = 101;          //array[1] = 101
DM[8'h1A] = 102;          //array[2] = 102
DM[8'h1B] = 103;          //array[3] = 103
DM[8'h1C] = 104;          //array[4] = 104
DM[8'h1D] = 105;          //array[5] = 105
DM[8'h1E] = 106;          //array[6] = 106
DM[8'h1F] = 107;          //array[7] = 107
end

//---- Read instruction memory (IM) -----
always@(*)
begin
    opcode <= IM[imAddress][15:8]; //ImAddress is instruction word address
    operand[7:0] <= IM[imAddress][7:0];
    operand[15:8] <= {8{operand[7]}}; //sign extend 8-bit operand to create a 16-bit
                                     //operand
end

//----- Decode and Execute -----
always@(*)
begin
    case (opcode)
        NOP: result = 16'hx;
        ADD_I: result = acc + operand;
        ADD_D: result = acc + DM[dmAddress];
        CMP_I: if (acc > operand)
            result = {15'hx, 1'b1};
            else
            result = {15'hx, 1'b0};
        JGT: result = 16'hx;
        JMP: result = 16'hx;
        LD_I: result = operand;
        LD_D: result = DM[dmAddress];
        LD_X: result = DM[dmAddress + x];
        MVX: result = acc;
        ST: result = acc;
        default: result = 16'hx;
    endcase
end

//----- Fetch and Write Back -----
always@(posedge clock or posedge reset)
begin
    if (reset == 1)
    begin
        pp <= 0;
        acc <= 0;
        x <= 0;
        sr <= 0;
    end
    else

        case (opcode)
            NOP: pp <= pp + 2; //incrementing in bytes not in words
            ADD_I: begin acc <= result; pp <= pp + 2; end
            ADD_D: begin acc <= result; pp <= pp + 2; end
            CMP_I: begin sr <= result[0]; pp <= pp + 2; end
            JGT: if (sr == 1)
                pp <= operand[5:0];
                else
                pp <= pp + 2; //incrementing in bytes not in words
            JMP: pp <= operand[5:0];
        endcase
    end
end

```

```
LD_I: begin acc <= result; pp <= pp + 2; end
LD_D:  begin acc <= result;    pp <= pp + 2; end
LD_X:  begin acc <= result; pp <= pp + 2; end
MVX:   begin x <= result[4:0]; pp <= pp + 2; end //indexing 16-bit

words

      ST:   begin DM[dmAddress] <= result;  pp <= pp + 2; end
      endcase
    end
  endmodule
```

图 8-8 说明了在异步重启后程序开始执行部分的模拟波形。波形列出了十六进制形式的指令和十进制形式的寄存器 ACC、X 和 SR 的内容。图 8-9 说明了当 SR 位变为 1 时，模拟波形到达末尾。正如所预期的那样，图中显示 ACC = 828，为数组 array 中各元素之和。随着程序最后的“JMP 0x1A”(0x51A)跳转指令的执行，程序将一直在循环中执行。

326
2
227

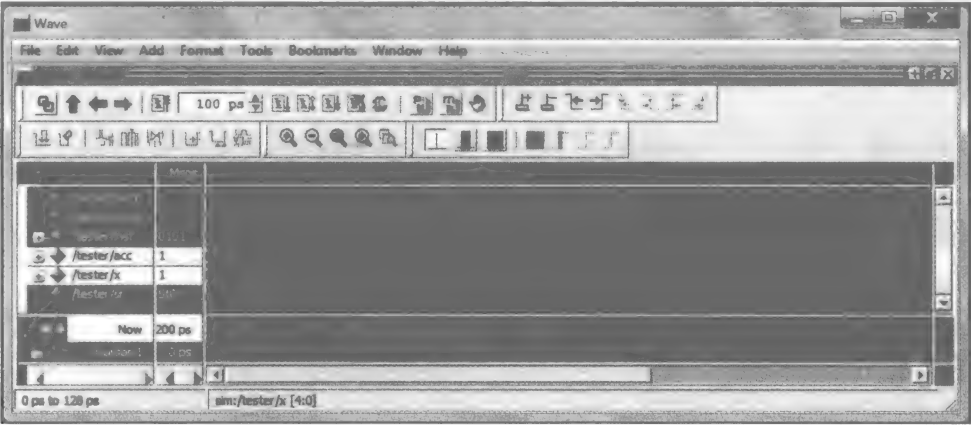


图 8-8 示例累加器 ISA 单周期处理器仿真波形，配图为波形的开始部分

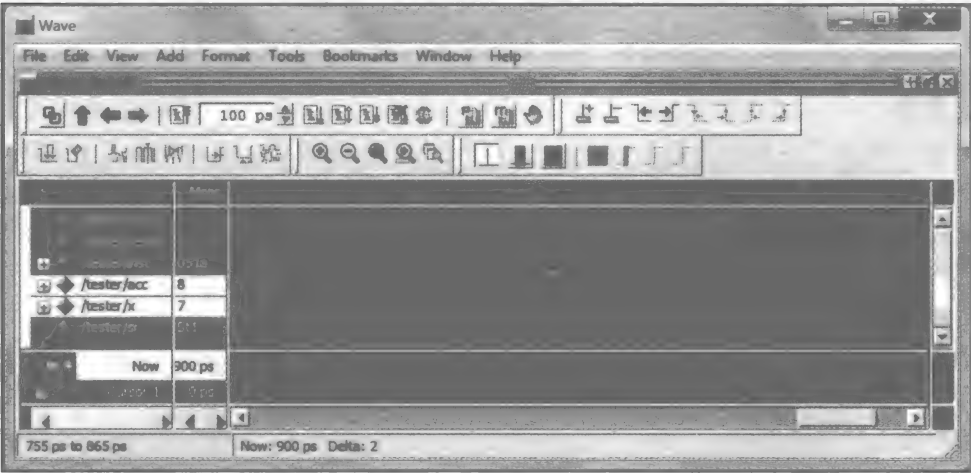


图 8-9 示例累加器 ISA 单周期处理器仿真波形，配图说明了波形的末尾部分，最终结果 sum 为 828 ($\sum_{i=0}^{107} i$)

8.3.3 累加器 ISA 处理器：流水线

图 8-10 说明了采用 4 级指令流水的示例累加器 ISA 的框图，图 8-11 说明了其细化的电

328 路。所有执行指令的控制信号在译码阶段生成，伴随着寄存器的内容，提供给执行阶段。写回阶段所需要的控制信号从执行阶段中发出，并伴随着计算的结果。写回阶段再选择并传输一个计算好的结果（如果有）到译码阶段，下一顺序的指令执行时可能用到当前执行的结果，写回阶段也将选中的结果转发给执行部件。

控制信号 ex 、 esr 以及来自写回阶段的 $eacc$ 信号在译码阶段控制将一个新的计算结果存储到寄存器 ACC 、 X 或 SR 中，这些信号在执行阶段也用来选择一个新计算结果。这需要三个额外的复用器来实现，这三个复用器标记为 $MUX5$ 、 $MUX6$ 、 $MUX7$ 。复用器使用控制信号 ex 、 esr 和来自写回阶段的 $eacc$ 信号构成了一个转发单元，并增加了流水线的吞吐量。如图所示，每一个复用器从寄存器的当前内容和一个从写回阶段反馈的还没有存入寄存器的新计算结果两者中选择。这使得处理器在执行数据相关的指令时，可直接使用从写回阶段返回的新的计算结果，从而不用等待计算结果写回到寄存器中。上述的转发单元的复杂性依赖于具体的 ISA 的复杂性，转发单元的功能在本节后面详细讨论。

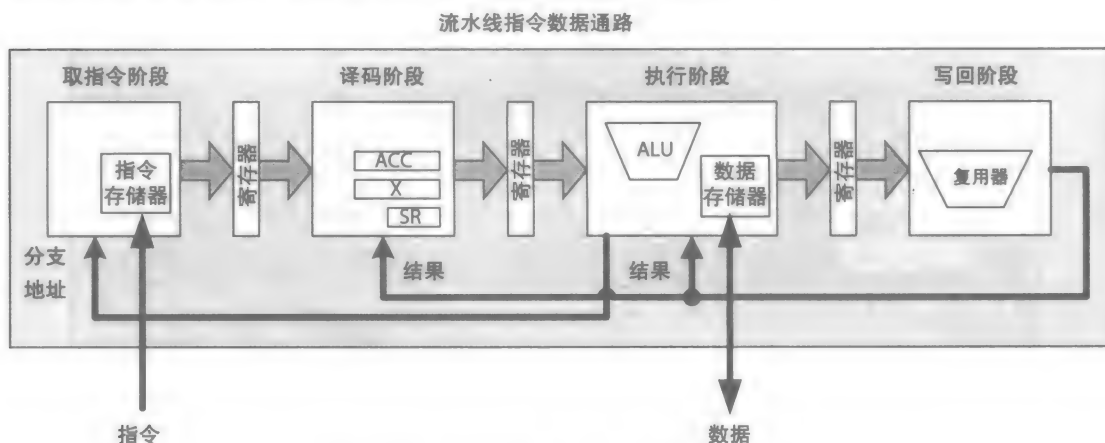


图 8-10 采用 4 级指令流水的数据通路

1. 仿真

图 8-12 说明了在异步重启后程序执行开始部分的仿真波形。波形列出了十六进制形式的指令和十进制形式的寄存器 ACC 、 X 和 SR 的内容。虽然波形显示指令正从一个阶段转到下一个阶段，但是从一个阶段到下一阶段传送的信息却与之前不同。写回阶段也被称为退出阶段。它表明指令执行的完成。当流水线中所有部件都处于忙状态并执行指令时，流水线的效率达到 100%。

图 8-13 说明当 SR 的某 1 位内容变为 1 时仿真波形的末端部分。注意每次跳转指令（“ JMP ”或“ JGT ”）执行时，流水线重新开始，这叫作流水线清空。例如，如图 8-13 所示，当无条件跳转指令 $0x0506$ （“ $JMP\ 6$ ”）或有条件跳转指令 $0x041A$ （“ $JGT\ 0x1A$ ”）执行跳转时，将导致流水线清空并因此降低流水线的效率。在上述示例中，当有流水线清空操作时，三个时钟周期内没有指令退出，如图中写回阶段（ $inst.WB$ ）所示。图中显示 ACC 内容为 828，为数组 $array$ 各元素之和。随着程序最后的“ $JMP\ 0x1A$ ”跳转指令，程序继续在循环中执行。

2. 转发和冒险单元

转发单元转发一个在写回阶段新生成的结果到执行阶段，在之前关于累加器 ISA 流水线处理器的示例中对转发单元进行了简要讨论。这里，我们将对其进行更全面的讨论。考虑

下面例 8-2 累加器 ISA 程序示例中的两条数据相关指令：

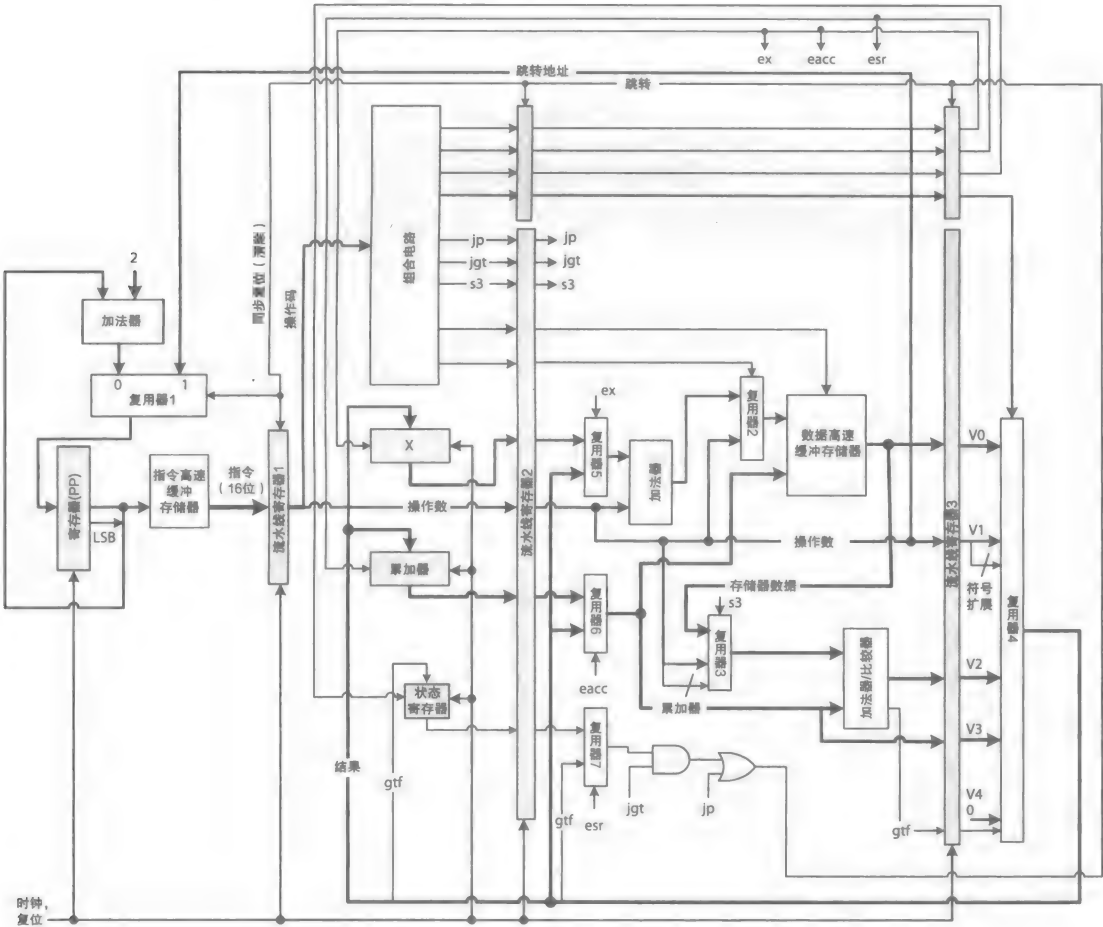


图 8-11 示例累加器 ISA 处理器 .vsd 的流水线数据通路

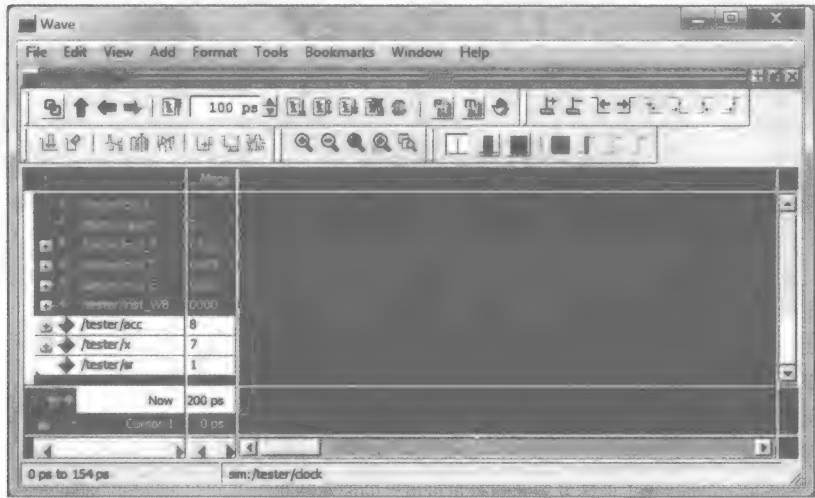


图 8-12 示例累加器 ISA 流水线数据通路的仿真波形；图中阐述了波形开始部分，各阶段从上到下显示



图 8-13 示例累加器 ISA 流水线数据通路的仿真波形；图中阐述了最终结果 sum 为 828 的波形结束部分

```
LD (i) //ACC ← M[i]
ADD 1 //ACC ← ACC + 1
```

LD 指令加载存储器数据到 ACC 中，ADD 指令继而递增 ACC 中的内容。图 8-14 展示了顺序执行着两个指令的流水线时空图。在图 8-14a 中，处理器使用一个转发单元转发一个还没有存储在 ACC 中的新读入的存储器数据（M[i]）到执行阶段中，以方便 ADD 指令执行时使用。

另一方面，图 8-14b 说明了不使用转发单元时执行同样两条指令的情况。在这种情况下，当 LD 指令处于执行阶段时，ADD 指令将进入译码阶段。当 LD 指令进入写回阶段时，ADD 指令的执行必须等到 ACC 指令更新完 M[i] 中的内容后才能进入执行阶段。因此，ADD 指令必须等待两个时钟周期，如图中冒泡所示。

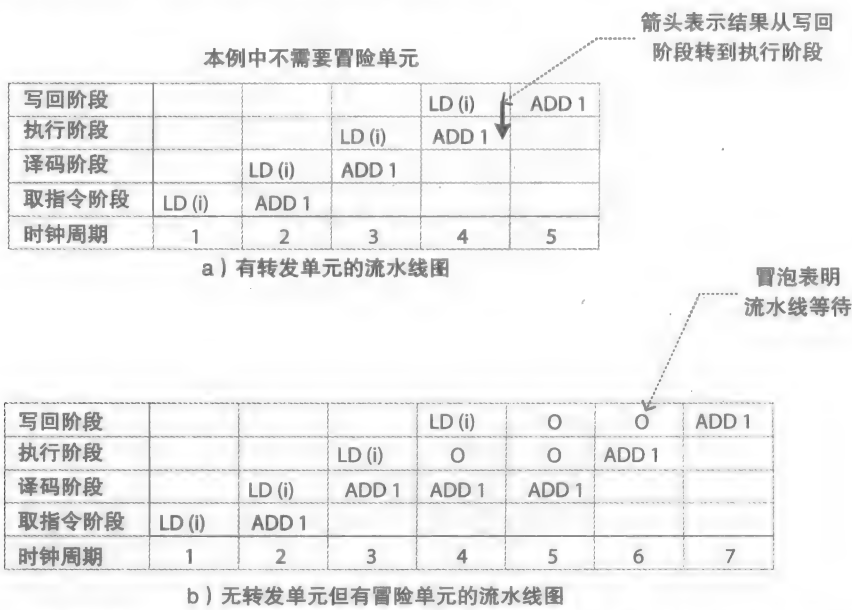


图 8-14 转发单元的作用：a) 有转发单元的流水线图；b) 无转发单元但有冒险单元的流水线图

流水线等待表明存在**冒险单元**，通过在该时钟周期隐含执行一个 NOP 指令来延迟或阻止下一条数据相关的指令（如 ADD）进入执行阶段。一旦 ACC 指令完成新数据的更新，数据相关指令（如 ADD）将被允许进入执行阶段完成执行过程。

因为在这种情况下执行单元仅包含一个执行部件（如图 8-14 中“执行”），处理器将需要一个转发单元或冒险单元，但并不是都需要。图 8-15 说明了使用冒险单元的累加器 ISA 流水线数据通路，例 8-9 展示了冒险单元的 HDL 模型。图 8-11 中用于实现转发单元的 MUX4、MUX5、MUX6 在图 8-15 中已从数据通路中移除。

例 8-9 HDL 代码部分使用显式声明的寄存器控制信号名称描述了图 8-15 中累加器 ISA 流水线处理器的冒险单元。冒险单元对写回阶段和执行阶段中的寄存器控制信号 *ex*、*eacc* 和 *esr* 进行比较。如果每对控制信号（如执行阶段中的 *eacc* 信号和写回阶段中的 *eacc* 信号）是相同的，位于译码阶段中的冒险单元将延迟当前指令的执行。特别的，冒险单元将维护 hazard 信号并同步复位（清除）流水线寄存器，流水线寄存器传递从译码阶段到执行阶段的当前指令的寄存器控制信号，如图所示。

331
?
332

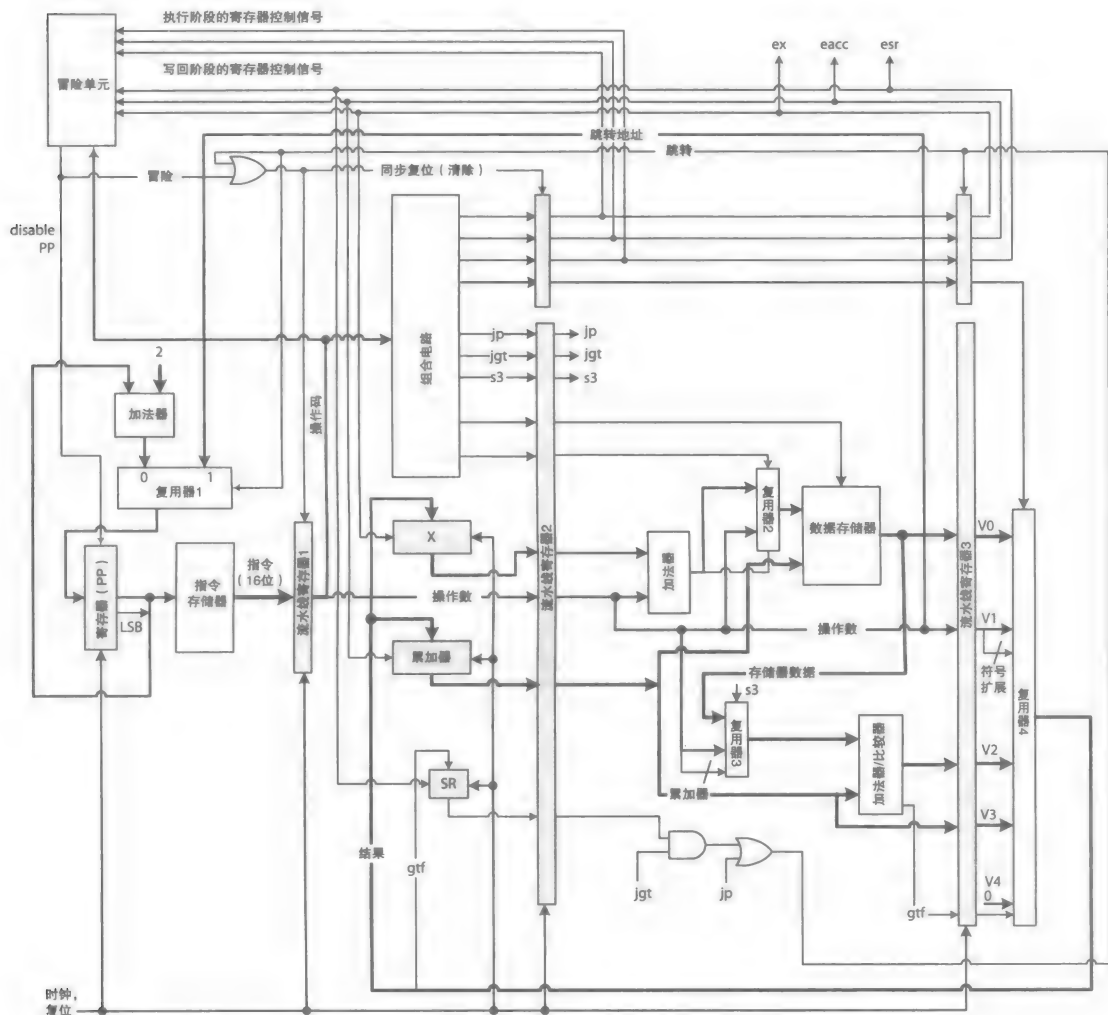


图 8-15 包含冒险单元的累加器 ISA 流水线数据通路示例

```

always@(*) //Hazard Unit
begin
    case(opcode)
    0: begin //NOP
        hazard = 0;
        end
    1: begin //ADD immediate
        if(eaccWB == 1 || eaccExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    2: begin //ADD direct
        if(eaccWB == 1 || eaccExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    3: begin //CMP immediate
        if(eaccWB == 1 || eaccExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    4: begin //JGT immediate
        if(esrWB == 1 || esrExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    5: begin //JMP address: PC <- address
        hazard = 0;
        end
    6: begin //LD immediate
        hazard = 0;
        end
    7: begin //LD direct
        hazard = 0;
        end
    8: begin //LD indexed
        if(exWB == 1 || exExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    9: begin //MVX
        if(exWB == 1 || exExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    10: begin //ST
        if(eaccWB == 1 || eaccExe == 1)
            hazard = 1;
        else
            hazard = 0;
        end
    default: begin
        hazard = 0;
        end
    endcase
end

```

一般情况下，执行单元由两个或多个阶段构成，从而将所需的硬件更好地分布到多个阶段中，并尽量减少了流水线的时钟周期。在这种情况下，流水线数据通路既需要一个转发单元也需要一个冒险单元，如将在 8.3.4 节中进行讨论的 RISC 处理器。此外，由于在一个典型的 CISC 或 RISC ISA 中使用不是一个而是多个通用寄存器，转发单元和冒险单元必须能够检查几个不同的寄存器中的数据相关。

3. 性能分析

一个处理器的性能通常以**每条指令的时钟周期数 (CPI)**来衡量。CPI 是程序执行总的时钟周期数除以程序中指令条数计算得来 (见公式 (8-1))。

$$\text{CPI} = \frac{\text{使用的时钟周期数 } (N)}{\text{程序执行的指令条数 } (n)} \quad (8-1)$$

图 8-16 说明了累加器 ISA 程序示例的流水线图, 如图所示。表 8-4 对流水线图进行了总结, 需要 6 个时钟周期 (图中标记为 i 到 vi) 来执行 for 循环之前的指令, 需要 12 个时钟周期来执行 for 循环中的每次循环的指令。最后, 需要 2 个时钟周期 (标记为 I 和 II) 来退出 for 循环操作。程序中 for 循环之前有 3 条指令, for 循环主体有 10 条指令, 需要 2 条指令 (“CMP” 和 “JGT”) 来退出 for 循环。

| | I | II | III | IV | V | VI | 1 | 2 | 3 | 4 |
|------|-----------|----------|----------|----------|-------------|-------------|-------------|-------------|-------------|-------------|
| 写回 | | | | LD 0 | ST (sum) | ST (I) | CMP 7 | JGT L2 | MOVX | LD X(array) |
| 执行 | | | LD 0 | ST (sum) | ST (I) | CMP 7 | JGT L2 | MOVX | LD X(array) | ADD (sum) |
| 译码 | | LD 0 | ST (sum) | ST (I) | CMP 7 | JGT L2 | MOVX | LD X(array) | ADD (sum) | ST (sum) |
| 取指令 | LD 0 | ST (sum) | ST (I) | CMP 7 | JGT L2 | MOVX | LD X(array) | ADD (sum) | ST (sum) | LD (I) |
| 时钟周期 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 写回 | 5 | 6 | 7 | 7 | 8 | 10 | 11 | 12 | 1 | 2 |
| 执行 | ADD (sum) | ST (sum) | LD (I) | ADD 1 | ST (I) | JMP L1 | O | O | CMP 7 | ... |
| 译码 | ST (sum) | LD (I) | ADD 1 | ST (I) | JMP L1 | O | O | CMP 7 | JGT L2 | ... |
| 取指令 | LD (I) | ADD 1 | ST (I) | JMP L1 | ? | O | CMP 7 | JGT L2 | MOVX | ... |
| 时钟周期 | ADD 1 | ST (I) | JMP L1 | ? | ? | CMP 7 | JGT L2 | MOVX | LD X(array) | ... |
| | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | ... |
| 写回 | ... | 10 | 11 | 12 | I | II | | | | |
| 执行 | ... | JMP L1 | O | O | CMP 7 | JGT L2 | O | O | | |
| 译码 | ... | O | O | CMP 7 | JGT L2 | MOVX | O | | | |
| 取指令 | ... | O | CMP 7 | JGT L2 | MOVX | LD X(array) | | | | |
| 时钟周期 | ... | CMP 7 | JGT L2 | MOVX | LD X(array) | ? | | | | |
| | ... | 100 | 101 | 102 | 103 | 104 | | | | |

图 8-16 例 8-2 中累加器 ISA 程序执行的流水线图

表 8-4 图 8-16 流水线图中的数据

| | |
|--------------------------|----|
| 执行 for 循环之前的指令的时钟周期数 | 6 |
| 执行 for 循环中的每次迭代所需要的时钟周期数 | 12 |
| 结束 for 循环的时钟周期数 | 2 |
| for 循环之前的指令条数 | 3 |
| for 循环主体中的指令数量 | 10 |
| 退出 for 循环的指令条数 | 2 |

公式(8-2)中, N 表示程序执行的总的时钟周期数, n 表示程序for循环进行 m 次执行的总的指令条数。

$$\begin{aligned}
 N &= 6 + m(12) + 2 \\
 &= 12m + 8 \quad \text{时钟周期} \\
 n &= 3 + m(10) + 2 \\
 &= 10m + 5 \quad \text{指令}
 \end{aligned}
 \tag{8-2}$$

当 $m = 8$ 时, 程序示例中的 CPI 计算如下:

$$\begin{aligned}
 \text{CPI} &= \frac{12m + 8}{10m + 5} \quad m \text{ 次迭代} \\
 &= \frac{12(8) + 8}{10(8) + 5} = \frac{104}{85} = 1.22 \quad m = 8
 \end{aligned}
 \tag{8-3}$$

公式 (8-4) 表明了当程序示例中 m 趋向于无穷大时的 CPI 值。由于执行 m 次 “JMP” 指令和一次 “JGT” 指令会导致跳转, 每次跳转都会清空流水线, 故而 m 趋向于无穷大时 CPI 大于 1。然而, 即使单周期处理器的 CPI 为 1, 单周期的数据通路相对于等价的流水线数据通路还是需要更长的时钟周期, 正如第 6 章中所述。

$$\text{CPI} = \frac{12m + 8}{10m + 5} = 1.2 \quad m \rightarrow \infty
 \tag{8-4}$$

在公式 8-4 中, CPI 为下限 (最小值)。如果一个程序的 CPI 为 1.0, 相应的流水线图没有冒泡, 表明流水线效率为 100%。CPI 的最小值为 1.2, 这个略大于 1.0 的数, 表明流水线的效率小于 100%, 并且, 在最优情况下, 平均每条指令的执行需 1.2 个时钟周期。CPI 值可用来估计 n 条指令的执行时间, 如公式 (8-5) 所示, 等式中 τ 表示每个时钟周期的秒数。

$$\text{执行时间} = n * \text{CPI} * \tau
 \tag{8-5}$$

由于指令间的数据相关、程序中的分支指令以及从高速缓冲存储器和存储器中存取数据时的延时的存在, 实现 CPI 为 1.0 是一件非常困难的任务。然而, 转发单元可通过解决某些指令间的数据相关性, 从而提升程序效率, 其他关于提升 CPI 的技术将在之后讨论。

8.3.4 RISC-ISA 处理器

图 8-17 说明了采用 5 级指令流水的 RISC-ISA 框图。DM (数据高速缓冲存储器) 现在被放在一个独立的阶段。这和最初的 MIPS 处理器的 5 级数据通路相似。执行阶段包括所有的算术操作和存储地址计算, 但不包括从高速缓冲存储器中访问数据, 从数据高速缓冲存储器中访问数据在 DM 阶段执行。5 级数据通路更适合 RISC-ISA, 因为数据项在用于执行算术指令前必须先加载到寄存器中。然而, 累加器 ISA 示例中所有的包含直接寻址 (D) 和变址寻址 (X) 的指令都必须转换为寄存器 - 寄存器的算术指令。这一数据通路具有简化流水线每一阶段的复杂性并实现阶段间传输时延更均匀的优点。

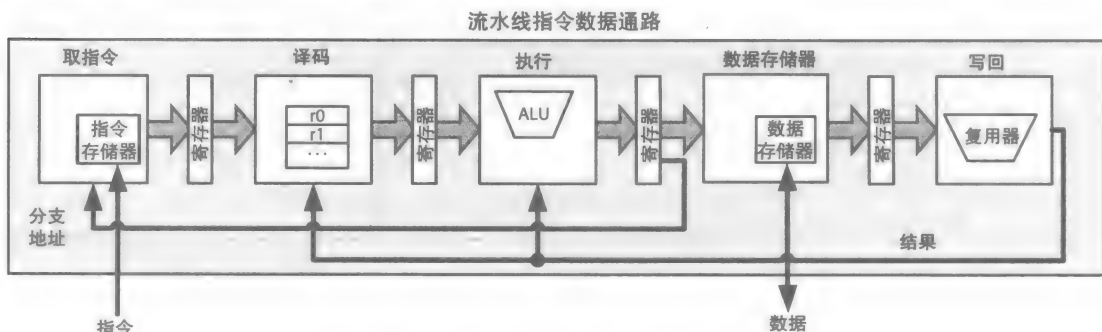


图 8-17 5 级 RISC-ISA 流水线数据通路

1. 程序示例

例 8-10 展示了与例 8-1 中高级语言程序相对应的 RISC-ISA 汇编程序。程序中假定寄存器 R0 始终为 0，寄存器 R1 到 R4 为通用寄存器，但并未执行代码优化。

例 8-10 与例 8-1 中程序相对应的 RISC-ISA 汇编程序；未进行编译器优化。

```
.code //start program code section

    ST      (sum), R0      //Initialize: M[sum] ← R0
    ST      (i), R0        //M[i] ← R0

L1:

    LD      R1, (i)        //R1 ← M[i]
    CMP     R1, 7          //is i > 7?
    JGT     L2             //exit for-loop if greater (PP ← L2)
    LD      R2, (sum)      //load sum, R2 ← M[sum]
    LD      R3, R1, (array) //load next array element, R3 ←
                          //M[array + R1]
    ADD     R4, R2, R3     //compute sum + array[i]
    ST      (sum), R4      //store sum, M[sum] ← R4
    ADD     R1, R1, 1      //increment i
    ST      (i), R1        //store i (M[i] ← R1).
    JMP     L1             //loop back

L2:    Ix                  //some instructions x, y, and z

    Iy

    Iz

.data //start program data section
array:  RB    8            //reserve 8 bytes
i:      RB    1            //reserve 1 byte
sum:    RB    1            //reserve 1 byte
```

337

2. 编译器优化

因为 DM 高速缓冲存储器在一个独立的阶段中，RISC-ISA 数据通路必须包含一个转发单元和一个冒险单元。这是因为“LD R3, R1, (array)”和“ADD R4, R2, R3”指令是数据相关的，因此需要 1 个时钟周期的延时来更新 R3 中的内容，从 DM 中读出，转发到执行阶段，如图 8-18 所示。当指令“LD R3, R1, (array)”进入执行阶段时，指令“ADD R4, R2, R3”处于译码阶段，当“LD”指令进入 DM 阶段时，冒险单元必须阻止“ADD”指令进入执行阶段，这是通过一个冒泡（一个隐含的 NOP 指令）来实现的，如图所示。

然而，因为存储器 load 指令的存在，编译器有可能通过优化程序从而消除冒泡，正如图 8-18 中的例子。在这个示例中，编译器可调整程序执行顺序，通过延迟与 LD 指令相关的某些或全部指令的执行，从而消除冒泡。例 8-11 展示了一个优化过的代码，将例 8-10 中的指令“ADD R1, R1, 1”移动到指令“LD R3, R1, (array)”和指令“ADD R4, R2, R3”之间，指令“ADD R4, R2, R3”的执行延时了一个时钟周期，从而消除了一个冒泡。

例 8-11 对例 8-10 中程序进行编译器优化，指令“ADD R4, R2, R3”的执行延时了一个时钟周期：

```
.code //start program code section

    ST      (sum), R0      //Initialize: M[sum] ← R0
    ST      (i), R0        //M[i] ← R0
```

```

L1:
    LD    R1, (i)          //R1 ← M[i]
    CMP   R1, 7            //is i > 7?
    JGT   L2              //exit for-loop if greater (PP←L2)
    LD    R2, (sum)        // load sum, R2←M[sum]
    LD    R3, R1, (array)  //load next array element, R3←
                          //M[array + R1]
    ADD   R1, R1, 1        // increment i; instruction moved
here
    ADD   R4, R2, R3       // compute sum + array[i]
    ST    (sum), R4        // store sum, M[sum]←R4
    ST    (i), R1          // store i (M[i]←R1).
    JMP   L1              // loop back
L2:     Ix                // some instructions x, y, and z
        Iy
        ...Iz

.data   //start program data section
array:  RB    8           // reserve 8 bytes
i:      RB    1           // reserve 1 byte
sum:    RB    1           // reserve 1

```

编译优化工作必须小心操作以避免错误修改程序。由于超出边界的数据访问可能会导致程序生成非法的输出，甚至执行失败。如例 8-10 中，移动指令“LD R2, (sum)”和“LD R3, R1, (array)”到“JGT”指令前并不是消除“ADD R4, R2, R3”指令执行所需要的单周期延时的正确的编译器优化步骤。原因是当“LD R3, R1, (array)”指令放置在“JGT”之前时，最后一次循环后，“LD R3, R1, (array)”指令将试图访问 array[8]，array[8] 是超出数组边界的元素，例 8-1 中数组 array 只含有 8 个元素 array[0] 到 array[7]。

3. 性能分析

图 8-17 中 RISC 数据通路包含 5 个流水线阶段，相对的，图 8-10 中与其等价的累加器 ISA 流水线数据通路包含 4 个流水线阶段。累加器 ISA 数据通路的执行阶段包含数据高速缓冲存储器，因而其传输延时最长，将 RISC 数据通路中该阶段分为两个阶段（执行阶段和 DM 阶段），使得每个阶段的传输延时更小。通过降低流水线阶段中最长传输时延的方式，使得处理器可以使用更快的时钟，从而提升了处理器的吞吐量。

此外，如例 8-11 所述，通过编译器优化，使用另一条指令（如“ADD R1, R1, 1”）的执行来代替存储器访问必需的一个时钟周期延时，从而进一步提升程序的 RISC CPI，这一方式是有可能的。指令“ADD R1, R1, 1”和指令“LD R3, R1, (array)”是数据相关的，因此，将指令“ADD R1, R1, 1”从例 8-10 中的位置移动到例 8-11 中的指令“ADD R4, R2, R3”之前并不会改变最初程序执行的正确性——只是指令执行的顺序变了。指令“ADD R1, R1, 1”更早执行，“ADD R4, R2, R3”指令延后一个时钟周期执行从而提供时间给“LD R3, R1, (array)”指令从 DM 中加载数据。更新过的数据将转发到执行阶段，供“ADD R4, R2, R3”指令执行使用。

正如将在第 10 章中所讨论的，如果目标数据并不在 DM（一个高速缓冲存储器）中，程序需要从一个低级的高速存储缓冲器或主存中复制数据，这个过程需要多个 CPU 时钟周期。然而，在下面的章节中，我们将深入讨论当数据不在高速缓冲存储器中时如何使用多线程来提高流水线的效率。

| | | | | | | | |
|-------|------------|------------|----------------|----------------|----------------|----------------|----------------|
| 写回 | ... | ... | ... | ... | LD R2, ... | LD R3, ... | O |
| 数据存储器 | ... | ... | ... | LD R2, ... | LD R3, ... | O | ADD R4, R2, R3 |
| 执行 | ... | ... | LD R2, ... | LD R3, ... | O | ADD R4, R2, R3 | ... |
| 译码 | ... | LD R2, ... | LD R3, ... | ADD R4, R2, R3 | ADD R4, R2, R3 | ... | ... |
| 取指令 | LD R2, ... | LD R3, ... | ADD R4, R2, R3 | ... | ... | ... | ... |

图 8-18 LD 指令和 ADD 指令之间的 RISC 数据相关

8.4 先进的处理器架构

RISC 流水线提高了指令吞吐量，但额外的性能提升将来自于减少流水线时钟周期。一种提升流水线性质的方法是将数据通路中的一个或多个流水线阶段划分为更小的子数据通路，从而产生一个拥有更多阶段的更深的流水线。因为每个更小的阶段会有一个较短的传播时延，更深的流水线将可使用更快的时钟，从而提高指令吞吐量。

如果一个流水线阶段的数据通路物理上不可分（例如，包含一个存储器），流水线阶段可被修改为包含不可分的硬件的两个或更多个副本，从而使得该阶段可分。所有的副本是同步并重叠操作的，在流水线阶段内并行运行，称之为超流水线。虽然超流水线阶段的传输时延保持和最初阶段的传输时延一致，但是，这个包含重复副本的流水线阶段可以更高频率生成输出结果。然而，深度流水线（包括超流水线）增加了功耗。

一方面，深度流水线提升了指令吞吐量，但是程序流控制指令（有条件的或无条件的）降低了流水线效率、提升了 CPI 值，因而降低了流水线吞吐量。现代处理器使用分支预测机制来尽量减少流水线清空操作，从而提升流水线效率。

此外，现代处理器使用指令级并行（ILP）来增加指令吞吐量。这种情况下，处理器属于超标量体系结构，因为处理器中每个阶段包含额外的资源并能够并行执行多个独立不相关的指令。对于一个程序来说，即使采用完美的分支预测机制，使用 ILP 的流水线的效率仍旧达不到 100%。这是因为，采用 k 条指令并行的流水线会根据指令之间的数据相关性执行一条、两条直到 k 条指令，甚至有时由于无法及时从高速缓冲存储器或主存储器中获取数据而根本不执行任何指令。因此，部分或全部数据通路中的资源在一个或多个时钟周期内保持空闲状态，除非流水线已经准备好同时执行多个程序（线程）。这种情况下，流水线也被称为使用了多线程。

340

在下面的几节中，我们将进一步探讨深度流水线、分支预测、静态或动态调度的 ILP 以及多线程。

8.4.1 深度流水线

图 8-19a 展示了初始的 4 阶段流水线，其中第 3 阶段传输时延最长，为 $2\Delta + \Delta_{\text{clocking}}$ 。其中 $\Delta_{\text{clocking}} = \tau_{\text{st}} + \tau_{\text{cq}} + \tau_{\text{cs}}$ ； τ_{st} 、 τ_{cq} 和 τ_{cs} 分别表示建立时间、时钟到 q 时间和时钟相位差。在图 8-19 b 中，第 3 阶段的硬件（如多级 MUX、CLA 加法器、组合除法等）被分为两个更小的模块，每个模块有更小的传输时延。

例如，为了简单起见，我们考虑第 3 阶段包含一个 2 级的 4-1 MUX，采用 3 个 2-1 MUX（第 3 章）设计。4-1 MUX 可被分割为两部分，并按两个阶段进行操作，这两个阶段中每个阶段的时延大约为图 8-19a 中第 3 阶段传输时延的一半。新的流水线有 5 个阶段，并使用更短的时钟周期 $\tau_{\text{new}} = \Delta + \Delta_{\text{clocking}}$ 进行操作。在这种情况下，唯一额外使用的硬件是流

水线寄存器。

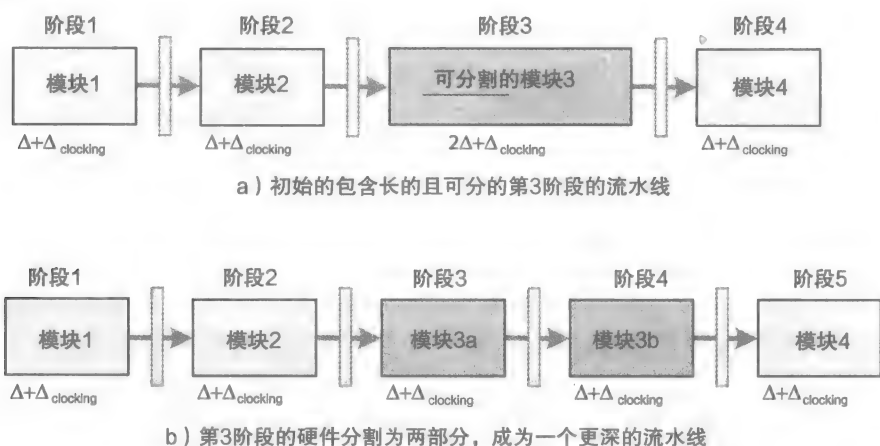


图 8-19 更深的流水线设计：a) 初始的第 3 阶段可分割的 4 阶段流水线；b) 初始的流水线转换为 5 阶段流水线，降低了时钟周期

另一方面，图 8-20a 展示了一个 4 阶段流水线，其中第 3 阶段的数据通路不可分割为更小的阶段。例如，假设图中的第 3 阶段为 DM 阶段。在这种情况下，存储器可被组织成两路低位交叉存储的方式（参见第 7 章），使得其奇地址的内容在一个模块中而偶地址的内容在另一个模块中。只要存储器连续交替地从奇地址和偶地址中访问数据，对两个交叉存储器的访问操作可重叠，生成超流水线，如图 8-20b 中的两个模块所示。总之，任何流水线阶段都是可以采用超流水线技术的。

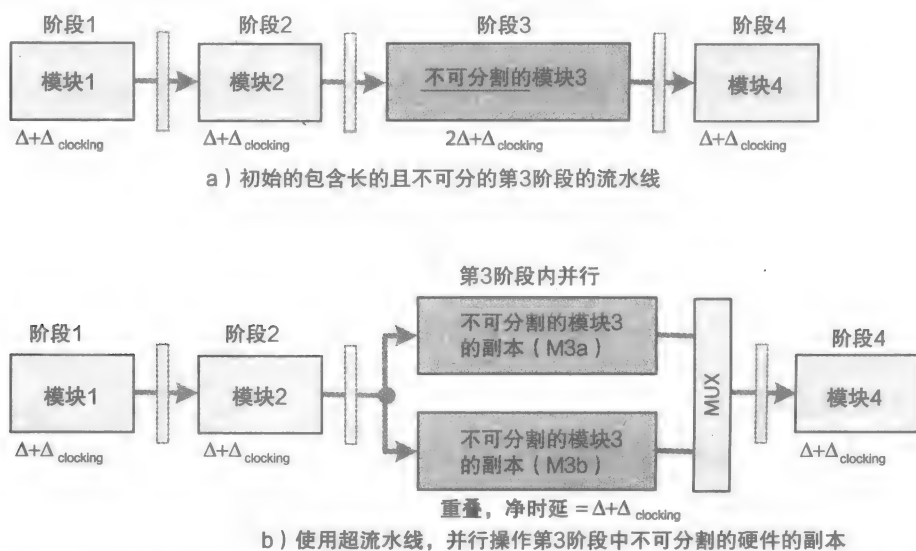


图 8-20 超流水线设计：a) 第 3 阶段不可分的 4 阶段流水线；b) 第 3 阶段采用超流水线

表 8-5 说明了使用图 8-20b 中的数据通路的超流水线过程，第 3 阶段中两个完全一样的不可分割模块分别标记为 M3a、M3b。当时钟周期 $\tau = \Delta + \Delta_{\text{clocking}}$ 时，在时间为 3τ 时，指令 I1 进入第 3 阶段，使用 M3a 执行，需 2 个时钟周期或约 2τ 来完成。注意 Δ 也包含 MUX

的延时。

因为流水线控制器轮换交替使用 M3a 和 M3b，一个时钟周期后，即时间为 4τ 时，当 I2 进入第 3 阶段时，流水线控制器将选择 M3b，这也需要 2τ 时间来完成。因此，虽然 M3a 和 M3b 每个都需要两个时钟周期来产生一个结果，但每个时钟周期第 3 阶段都会有结果输出。

表 8-5 表中说明了在图 8-20b 中的超流水线第 3 阶段，其采用了两个相同模块（标记为 M3a、M3b）

| | | | | | | | |
|----------|---------|---------|--------------------|--------------------|--------------------|--------------------|-----|
| 阶段 4（写回） | | | | O^a | I1 | I2 | I3 |
| 阶段 3（执行） | | | I1 (M3a, 2τ) | I2 (M3b, 2τ) | I3 (M3a, 2τ) | I4 (M3b, 2τ) | ... |
| 阶段 2（译码） | | I1 | I2 | I3 | I4 | I5 | ... |
| 阶段 1（取指） | I1 | I2 | I3 | I4 | I5 | I6 | ... |
| 时间 | 1τ | 2τ | 3τ | 4τ | 5τ | 6τ | ... |

$\&$ 因为超流水线产生的冒泡。

341
?
342

当时间为 5τ 时，第 3 阶段 M3a 的输出将提供给第 4 阶段，允许指令 I1 继续执行。在时间为 6τ 时（下一个时钟周期），第 3 阶段 M3b 的输出将提供给第 4 阶段，允许指令 I2 继续执行。这使得流水线达到每个时钟周期 τ 执行一条指令的效率，如表中所示。

深度流水线技术使得流水线可采用更快的时钟来提升流水线性能。在图 8-19a 中，最初的时钟周期 $\tau_{old} = 2\Delta + \Delta_{clocking}$ ，在图 8-19b 中，时钟周期降低为 $\tau_{new} = \Delta + \Delta_{clocking}$ ，时钟频率接近原时钟的两倍。在图 8-20b 中，时钟频率也从之前的图 8-20a 中的 $2\Delta + \Delta_{clocking}$ 有效降低为接近 $\Delta + \Delta_{clocking}$ 。

然而，深度流水线（包括超流水线）增加了每次流水线重启时所需的冒泡数量。在图 8-20b 中，由于超流水线技术的使用，当流水线阶段数从最开始的图 8-20a 中的 4 级改为 5 级，在时间为 4τ 时，比之前多需要一个冒泡。深度流水线通过同时操作多条指令增加了流水线的并发性，但是关于指令流水线的深度有一定的限制条件，如下所示：

- 深度流水线（包含超流水线）不仅增加了硬件的数量，也提升了流水线的时钟频率。当流水线中的硬件数量和时钟频率提升时，流水线的功耗也相应的有所增加（第 6 章）。因此，流水线的深度有一定的限制。
- 因为流水线需要执行跳转 / 分支指令，深度流水线（包含超流水线）的深度超过特定的限制时，效果可能适得其反。任何关于指令流顺序的改变都会造成一次流水线清空操作，这种情况下，流水线会引入更多的冒泡并因此降低了流水线的效率和性能。

8.4.2 分支预测技术

分支预测指的是在执行一个有条件或无条件的指令之前，确定程序流的方向。流水线可以在更早的时间确定目标跳转 / 分支地址，在取指令阶段获取目标地址中的指令，从而在程序流有变化时，最小化流水线中冒泡的数量。

[注意，为了简单起见，此处的跳转指令和分支指令我们是相同对待的，但是，一般来说，这两种指令的分支预测的实现方式还是不一样的。分支地址通常的计算的结果是与当前 PP 中的指令地址的一段距离（位移），而跳转地址的计算结果是一个确定的地址（与 PP 中内容不相关）。一般而言，分支指令用于跳转距离比较小的情况，例如实现一个 for 循环；而跳转指令用于一个长距离的跳转，例如一个子函数调用。在下面的章节中，术语“跳转”和“分支”是可交换的，都会造成程序流的改变。]

343

在图 8-11 或图 8-17 中, 当当前指令为一个跳转时 (如 “JGT” 或 “JMP”), 程序流可以改变方向的最早时间是跳转指令进入写回阶段。例如, 在累加器 ISA 数据通路中, 当跳 “JMP” 令进入执行阶段 (如图 8-15 中的第 15 个时钟周期), 在下一个时钟周期, “JMP” 移动到写回阶段, PP 中的内容也改变为目的地址 L1, 取指令阶段转而获取指令 “CMP 7”。这将使得流水线进行清空操作, 丢弃所有部分执行 (处理) 的指令, 流水线继而从分支地址开始执行。然而, 通过使用额外的硬件, 分支方向可以被预测, 从而降低流水线的冒泡数量并提升流水线的性能。

1. 静态分支预测

静态或默认的分支预测可以自身使用或与动态分支预测结合使用。考虑一个分支指令, 如例 8-11 中的 “JGT L2”, 其中 L2>PP 为一个前向分支地址。在这种情况下, 只要 for 循环一直执行, 对于该指令采取 “不跳转” 作为默认分支决定将是正确的。只有当程序跳出 for 循环时, 处理器分支预测的方向才是错误的。

同样, 当条件指令造成后向分支, 例如, 当执行 do-while 时, 条件语句在循环的末尾, 对于该跳转指令, 只要 do-while 继续执行, 采取 “跳转” 作为默认分支决定都是正确的。只有当程序跳出 do-while 循环时, 处理器分支预测的方向才是错误的。

通常情况下, 静态预测的规则在程序编译优化过程中被考虑。例如, 例 8-1 中的跳转条件 “i<8” 将被编译为例 8-2 和例 8-11 中的 “CMP 7” 和 “BGT L2”。同样, 条件也会被编译成相似的指令, 如例 8-3 (奔腾处理器) 和例 8-4 (Sparc 处理器) 中所示。

静态分支预测可被执行的最早时间是当分支指令进入译码阶段并且其操作码已知。然而, 这将导致执行静态分支预测时必须有一个周期的延迟。例如, 如果流水线实现上述的静态分支预测 (即没有动态分支预测), 编译器可能会选择执行另一条指令从而覆盖这一个周期的延迟。

考虑例 8-11 中的无条件跳转指令 “JMP L1”。通过编译器优化操作来利用这一个周期的延迟是有可能的, 如例 8-12 中所示。为了利用这一个周期的延迟并在该周期执行一条有用的程序, 编译器可以从 for 循环主体中移动一条指令 (如 “ST (i), R1”) 到指令 “JMP L1” 之后。然而, 流水线必须被修改从而始终执行跟在无条件跳转 / 分支指令后的指令。

例 8-12 例 8-1 中的 RISC-ISA 汇编程序, 对指令 “JMP” 进行了静态分支预测的优化。将例 8-11 中的指令 “ST (i), R0, R1” 移动到指令 “JMP L1” 之后, 从而消除了指令 “JMP L1” 执行静态分支预测时所必需的一个周期的延迟。

```
.code //start program code section

    ST    (sum), R0          //Initialize: M[sum] ← R0
    ST    (i), R0           //M[i] ← R0

L1:

    LD    R1, (i)           //R1 ← M[i]
    CMP   R1, 7             //is i > 7?
    JGT   L2               //exit for-loop if greater (PP←

L2)

    LD    R2, (sum)         //load sum, R2 ← M[sum]
    LD    R3, R1, (array)   //load next array element, R3 ← M[array
+ R1]

    ADD   R1, R1, 1         //increment i; instruction moved here
    ADD   R4, R2, R3        //compute sum + array[i]
    ST    (sum), R4        //store sum, M[sum] ← R4
```

```
JMP  L1          //loop back
ST   (i), R1     //store i, M[i] ← R1;
                        //instruction moved here

L2:   Ix          //some instruction x

.data //start program data section
array: RB 8      //reserve 8 bytes
i:     RB 1      //reserve 1 byte
sum:   RB 1      //reserve 1 byte
```

在这些情况下，若编译器无法移动任何指令到无条件跳转 / 分支指令之后，编译器必须在无条件跳转 / 分支指令后插入一个 NOP 指令，如例 8-4 中未经优化的 AltraSparc II 程序中所示。注意，Sparc 程序在有条件跳转指令“gt”之后也插入了 NOP 指令。这是因为 Sparc 处理器也会执行跟在条件分支指令之后的指令。并且，当代码可进一步优化时，处理器也将提供给编译器进一步优化代码的选项。

2. 动态分支预测

当无条件跳转 / 分支指令（如“JMP”）每次都是跳转并改变程序流时，预测分支决定为“跳转”是很简单的。但是，当有条件跳转指令（如“JGT”）是数据相关时，预测“跳转”还是“不跳转”就比较难了，而且不会总是预测正确，尤其是当有条件分支指令控制的的是一个循环内的“if-else”语句时。这里的跳转和不跳转指的是根据指令语句中 if 条件的真假，执行之后的“then”或“else”部分的代码。通过在 for 循环和 do-while 语句使用后向分支预测技术，动态分支预测器可以更好地描述早期的工作。然而，好的前向分支预测对于提升流水线的效率也是极其必要的。

345

现代处理器使用动态分支预测机制，通过收集已经执行的有条件的或无条件分支指令的历史执行结果，对当前的分支指令进行预测。最初，当一个程序开始执行并且没有分支的历史数据时，将使用静态分支预测机制预测有条件 / 无条件跳转指令的执行。为了流水线性能的最优化，分支的历史数据和预测逻辑保存在取指令阶段。

在我们讨论动态分支预测技术之前，我们通过表 8-6 说明了执行例 8-11 中 for 循环中的两次迭代过程，但做了以下假设：

表 8-6 使用 RISC 流水线执行两次迭代过程

| 时钟周期 | 取指令 | 译 码 | 执 行 | 数据存储器 | 写 回 |
|-------|--------------------|--------------------|--------------------|------------------------|--------------------|
| 1 | ST (sum), R0 | | | | |
| 2 | ST (i), R0 | ST (sum), R0 | | | |
| 3: L1 | LD R1, (i) | ST (i), R0 | ST (sum), R0 | | |
| 4 | CMP R1, 7 | LD R1, (i) | ST (i), R0 | ST (sum), R0 | |
| 5 | JGT L2 | CMP R1, 7 | LD R1, (i) | ST (i), R0 | ST (sum), R0 |
| 6 | LD R2, (sum) | JGT L2 (静态: 不跳转) | CMP R1, 7 | LD R1, (i) | ST (i), R0 |
| 7 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 | CMP R1, 7 | LD R1, (i) |
| 8 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 (决定: 不跳转, 开始历史) | CMP R1, 7 |
| 9 | ADD R4, R2, R3 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 |
| 10 | ST (sum), R4 | ADD R4, R2, R3 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) |
| 11 | ST (i), R1 | ST (sum), R4 | ADD R4, R2, R3 | ADD R1, R1, 1 | LD R3, R1, (array) |
| 12 | JMP L1 | ST (i), R1 | ST (sum), R4 | ADD R4, R2, R3 | ADD R1, R1, 1 |

(续)

| 时钟周期 | 取指令 | 译 码 | 执 行 | 数据存储器 | 写 回 |
|--------|--------------------|--------------------|--------------------|---|--------------------|
| 13 | Ix | JMPL1 (静态: 跳转) | ST (i), R1 | ST (sum), R4 | ADD R4, R2, R3 |
| 14: L1 | LD R1, (i) | O | JMPL1 | ST (i), R1 | ST (sum), R4 |
| 15 | CMP R1, 7 | LD R1, (i) | O | JMPL1 (决定: 跳转, 开始历史) | ST (sum), R4 |
| 16 | JGT L2 | CMP R1, 7 | LD R1, (i) | O | JMPL1 |
| 17 | LD R2, (sum) | JGT L2 | CMP R1, 7 | LD R1, (i) | O |
| 18 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 | CMP R1, 7 | LD R1, (i) |
| 19 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 | CMP R1, 7 |
| 20 | ADD R4, R2, R3 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 |
| 21 | ST (sum), R4 | ADD R4, R2, R3 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) |
| 22 | ST (i), R1 | ST (sum), R4 | ADD R4, R2, R3 | ADD R1, R1, 1 | LD R3, R1, (array) |
| 23 | JMPL1 (动态: 跳转) | ST (i), R1 | ST (sum), R4 | ADD R4, R2, R3 | ADD R1, R1, 1 |
| 24: L1 | LD r1, (i) | JMPL1 | ST (i), R1 | ST (sum), R4 | ADD R4, R2, R3 |
| 25 | CMP R1, 7 | LD R1, (i) | JMPL1 | ST (i), R1 | ST (sum), R4 |
| 26 | JGT L2 (动态: 不跳转) | CMP R1, 7 | LD R1, (i) | JMPL1 (Decision: taken, update history) | ST (i), R1 |
| 27 | LD R2, (sum) | JGT L2 | CMP R1, 7 | LD R1, (i) | JMPL1 |
| 28 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 | CMP R1, 7 | LD R1, (i) |
| 29 | ADD R1, R1, 1 | LD R3, R1, (array) | LD R2, (sum) | JGT L2 (决定: 跳转, 更新历史) | CMP R1, 7 |
| 30: L2 | Ix | ADD R1, R1, 1 | LD R3, R1, (array) | O | JGT L2 |
| 31 | ? | Ix | ADD R1, R1, 1 | O | O |
| 32 | ? | ? | Ix | O | O |
| 33 | ? | ? | ? | Ix | O |
| 34 | ? | ? | ? | ? | Ix |

图 8-17 中的 RISC 流水线既实现了静态分支预测逻辑，也实现了动态分支预测逻辑。早先讨论的静态分支预测机制是在译码阶段实现的，而一个“完美”的动态分支预测机制是在取指令阶段实现的。注意，在这种情况下，编译器不需要利用完成静态分支预测所必需的一个周期的延迟来优化代码。

346
?
347

对于有条件跳转指令“JGT L2”，在执行的第 6 个时钟周期，静态分支预测决定为“不跳转”，程序连续执行下一顺序指令“LD R2, (sum)”。当第 9 个时钟周期指令“JGT L2”执行完时，动态分支预测机制也将更新分支预测决定，为“不跳转”。对于第 13 个时钟周期的无条件跳转指令“JMPL1”，动态分支预测的决定为“跳转”，将从跳转地址 L1 处的指令“LD R1, (i)”开始执行，并在第 14 个时钟周期产生一个冒泡。在第 16 个时钟周期，指令“JMPL1”退出流水线，动态预测机制也将更新分支预测决定为“跳转”。对于第 23 个时钟周期的指令“JMPL1”，动态分支预测机制预测为“跳转”，程序继续执行指令“LD R1, (i)”，并且不会在第 24 个时钟周期产生额外的流水线冒泡。

直到最后一次迭代之前，程序执行中都没有产生与分支相关的多余的冒泡。在第 29 个时钟周期最后一次执行时，由于执行跳转指令“JGT L2”导致预测错误。程序执行跳转导

致了流水线清空操作，引入了3个流水线冒泡，程序将继续执行 for 循环之后的“Ix”指令。

图 8-21 说明实现一个 1 位动态分支预测器的有限状态图 (FSD)。如表 8-6 所示，当指令“JGT L2”第一次执行时，因为 $L2 > PP$ 表明 L2 为前向跳转地址，译码阶段中的动态分支预测仪将正确地预测“不跳转”。第 8 个时钟周期的决定“不跳转”将在第 9 个时钟周期初始化 1 位的预测器，初始化为“预测不跳转”。对于剩余的迭代，预测器将准确地预测为“不跳转”。最后一次迭代中，预测器对“JGT L2”指令的执行（如：第 29 个时钟周期 for 循环的两个迭代）将预测错误，这也将导致 1 位的预测器的状态转变为“预测跳转”。

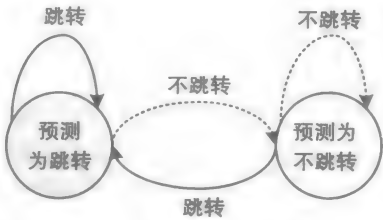


图 8-21 1 位动态分支预测器的 FSD

如果 for 循环只执行一次，1 位的动态预测器将如期望的那样预测正确，如表 8-6 所示。然而，假设例 8-11 中的 for 循环作为一个内部循环并执行多次。for 循环第二次执行时，1 位预测器的当前状态为“预测跳转”，这将导致 for 循环开始执行指令“JGT L2”时，预测错误，预测器的状态将转变为“预测不跳转”。1 位的预测器还将在 for 循环的最后预测错误一次。因此，1 位的预测器的执行效果要比单独的静态分支预测的效果差。

348

3. 2 位动态预测器

图 8-22 说明了 2 位动态预测器的有限状态图 [7]。我们继续以例 8-11 中执行的 for 循环为例。for 循环第一次执行时，关于指令“JGT L2”的 2 位动态预测器的有限状态图的状态将被初始化为“预测不跳转”。2 位动态预测器将一直保持“预测不跳转”状态，直到最后一次跳出 for 循环时预测错误，使得预测器的状态转为“预测可能不跳转”。

for 循环第二次执行时，处于“预测可能不跳转”状态的动态预测器将在指令“JGT L2”第一次迭代执行中预测正确，当指令“JGT L2”在第一次迭代中执行完成时，程序执行不跳转，这表明预测正确，使得 2 位有限状态图的状态正确地改变为“不跳转状态”，这样 for 循环在执行第一个迭代时将不预测错误。2 位动态预测器只会在 for 循环的末尾预测错误一次，正如期望的那样，每次 for 循环执行动态预测只错误预测一次。

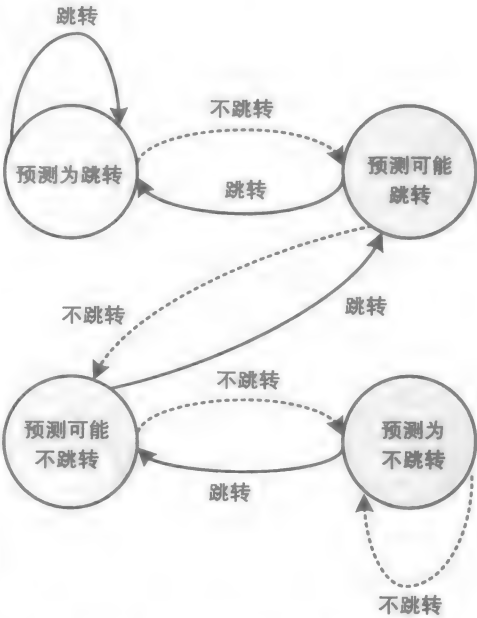


图 8-22 2 位动态分支预测算法 [7]

图 8-23 是在取指令阶段实现的 2 位动态分支预测器的数据通路。数据通路中包含了一个指令分支历史表 (BHT)，分支历史表用于保存每条分支指令的分支指令地址 (BIA)、分支目标地址 (BTA) 以及一个 2 位动态预测器的 2 位当前状态 (CS)。2 位的存储空间是由两个触发器构成，其用于存储表中每条指令的当前状态 (CS)，其也将用于实现有限状态机 (FSM)。分支历史表将根据表的大小保存最近刚执行的分支指令的相关历史。在一个程序执

4. 基于分支相关性的预测

对于“if-else”语句重复执行的情况（如在一个循环中），已经证明，对于其可能执行的数据通路的预测，分支预测算法的效果要比使用独立的2位预测器效果好[8，9]。例如，考虑下面的for循环中的3个if-else语句，并假定编译器将每一个“if-else”的条件语句翻译成有条件的分支指令，即分支决定“不跳转”是指“then”部分的“if-else”代码。

程序代码 [8]

if-else 语句的编译器输出

```
for (...)  
{  
    ...  
    If(x == 2)  
        x = 0;  
    If(y == 2)  
        y = 0;  
    If(x != y) {  
        ...  
    }  
}
```

```
for(...)  
{  
    ...  
    BNE ... // when x = 2, BNE does not branch  
    ...  
    BNE // when y = 2, BNE does not branch  
    ...  
    BEQ // when x ≠ y, BEQ does not branch  
    ...  
}
```

到达指令“BEQ”有4种可能的执行路径，如图8-24所示。注意，对于指令“BEQ”，分支决定取决于执行路径。当x和y均为2时，可以确定“x!=y”为假。因此，对于某些程序，最近执行的分支指令的分支决定之间经常有很强的相关性，此信息可用于实现一个更好的动态预测器。

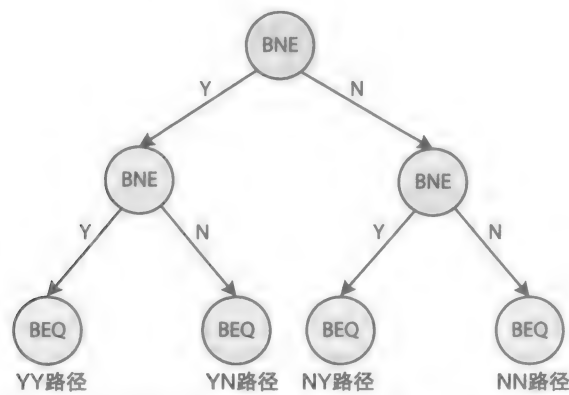


图 8-24 之前提到的在 for 循环中所有可能到达指令“BEQ”的执行路径

为了确定执行路径，分支预测寄存器（BPR）被用来对最近执行的分支决定进行编码。以前面的for循环为例，2位的BPR可编码图8-24中的4种执行路径为(11)₂、(10)₂、(01)₂和(00)₂，其中1表示是（Y），0表示否（N）。如果对于两条分支指令“BNE”的分支决策依次为“不跳转”（如x=2）和“跳转”（y≠2）时，相应的BPR中的内容为(01)₂。

图8-25说明了2位相关分支预测器的数据通路。在图中，PP的低位将和2位BPR中内容串联起来，生成一个BHT中的索引。根据图8-24中的程序采用的数据通路使用索引来决定针对指令“BEQ”选择不同的2位预测器。其他可参考练习章节的一些例子。

因为BPR确定了程序中的一个执行路径，这也就是说，BPR对程序执行提供了一个全

局视角。因为这个原因，基于相关性的预测器也经常被称为**全局预测**。还有其他的预测器结合本地和全局分支历史数据进行预测 [9]。

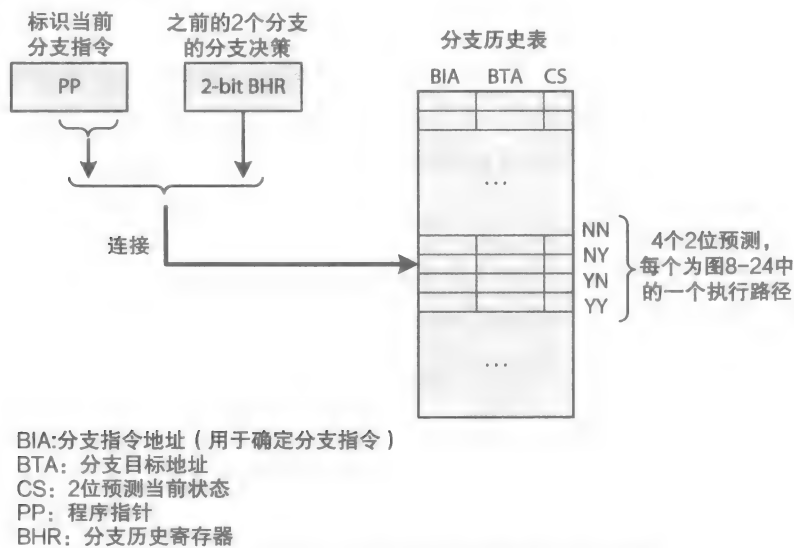


图 8-25 使用分支决定的全局视图的相关预测

352

对 SPEC89 基准测试程序的研究表明，错误预测的概率——例如，对于“gcc”编译器——当采用含有 4096 个条目的 BHT 进行的本地预测器（图 8-23）时为 12%，与之相对的是，当采用含有 1024 个条目但同样大小的 (1024*4 = 4096)BHT 进行的全局预测器（图 8-25）时为 11%。对于 Spice 电路仿真程序，本地预测错误率为 9%，全局预测错误率为 5%，对于 Espresso 逻辑最小化软件，本地预测为 5%，全局预测为 4%[10]。正如前面讨论的那样，这说明使用多种预测器比仅仅使用一种预测器的预测效果好。

5. 竞争预测器

现代处理器通常实现多种预测器，并对每条分支指令动态地选择效果最好的预测器。例如，考虑实现了一个本地预测器和一个全局预测器的预测机制，使用 2 位预测器（图 8-22）来选择最佳预测器。这一机制被称为**竞争预测器**，因为对于每条分支指令，经常选中的预测器（本地或全局）将是胜者。

例如，使用竞争预测器，对于 SPEC 整数基准测试程序，40% 的可能性将采用全局预测器，而对于 SPEC 浮点数基准测试程序，15% 的可能性将采用全局预测器 [10]。AMD 皓龙处理器和羿龙处理器都采用了竞争预测器。

8.4.3 指令级并行

为了在一个超标量处理器上并行执行两条或多条指令，这些指令必须是数据不相关的。当采用指令级并行时，有 4 种可能的依赖关系，如表 8-7 所示。无论 ILP 是否使用，程序中某些指令之间总是存在一定的数据依赖。另一方面，当指令并行执行（在同一流水线周期），两条指令间可能存在一个**反相相关性**和 / 或**输出相关性**。若一对指令之间存在一种或多种这样的相关性，当指令并行时，将导致**写后读 (RAW)**、**读后写 (WAR)**和**写后写 (WAW)**等冒险。并且，这些冒险不能在执行阶段消除。

表 8-7 ILP 中相关性类型

| 相关性类型 | 指令对示例 | 定 义 |
|-------|---------------------------------------|---|
| 数据相关性 | LD R1, (sum) ... ADD R3, R2, R1 | 一条指令（如“LD”）改变（写）一个寄存器（如 R1）中的内容，而另一指令（如“ADD”）使用（读）寄存器（R1）中的内容来执行其他计算，在同一流水线周期内执行这两条指令将导致 RAW 冒险 |
| 反相相关性 | ADD R3, R2, R1 ... LD R2, (sum) | 一条指令（如“ADD”）使用（读）了一个寄存器（如 R2）中的内容，一条后来执行的指令（如“LD”）改变了（写）该寄存器的内容。在同一流水线周期内执行这两条指令将导致 WAR 冒险 |
| 输出相关性 | ADD R3, R2, R1 ... LD R3, (sum) | 两条指令（如“ADD”和“LD”）改变了（写）同一寄存器（R3）的内容。同一流水线周期内执行这两条指令将导致 WAW 冒险 |

353

例如，考虑例 8-13 中在一个超标量处理器上执行的程序代码，指令之间存在数据相关性、反相相关性和输出相关性。指令并行执行时，将导致 RAW、WAR 和 WAW 冒险，如图 8-26 中图解所示。

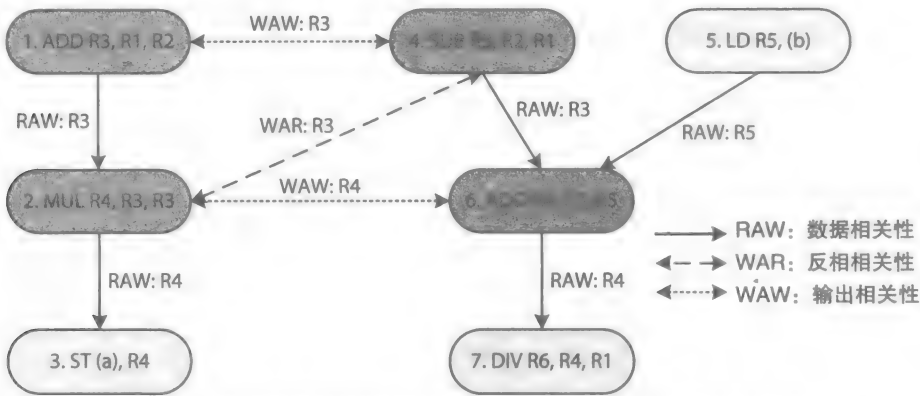


图 8-26 使用图解说明例 8-13 中指令间的 RAW、WAR 和 WAW

例 8-13 程序代码示例，假定寄存器 R1 和 R2 已被从存储器中读取的内容初始化：

```
1:  ADD R3,    R1, R2    //R3 ← R1 + R2;
2:  MUL R4,    R3, R3    //R4 ← R3 * R3;
3:  ST (a),    R4        //M[a] ← R4;
4:  SUB R3,    R2, R1    //R3 ← R2 - R1;
5:  LD R5,     (b)       //R5 ← M[b];
6:  ADD R4,    R3, R5    //R4 ← R3 + R5;
7:  DIV R6,    R4, R1    //R6 ← R4 / R1;
```

在示例程序中，存在 5 个 RAW、2 个 WAW 和 1 个 WAR 潜在的冒险。图中通过箭头（→）连接的一对指令之间是数据相关的，其不能并行执行。指令“ADD R3, R1, R2”和“SUB R3, R2, R1”之间有输出相关性，因为这两条指令都更新了寄存器 R3，这两条指令的并行执行将导致 WAW 冒险。诸如此类，指令“MUL R4, R3, R3”和指令“ADD R4, R3, R5”因为都向寄存器 R4 中写入数据，若这两条指令并行执行，将导致 WAW 冒险。指令“SUB R3, R2, R1”和“MUL R4, R3, R3”有反相相关性，MUL 的执行是基于指令“ADD R3, R1, R2”的结果，而不是“SUB”指令的结果，如果“SUB”指令在“MUL”指令之前执行，这将导致一个 WAR 冒险。

指令采用 ILP 的要求是同一流水线周期调度执行的指令之间没有数据相关性、反相相关

354 性和输出相关性。可以并行执行的指令在执行过程中由编译器静态选择（如软件结果）或者由硬件动态选择。软件和硬件解决方式的优点和缺点将在后面讨论。

1. 静态调度的 ILP

对于采用静态调度的 ILP，在每个流水线周期中，由编译器决定而不是由硬件来决定发射哪些指令来并行执行。一个超标量处理器的编译器必须检测一组指令并将其分为一个个集合。在同一集合中的指令之间必须没有任何会造成 RAW、WAR 或 WAW 的相关性。考虑例 8-13 中的程序代码。假定处理器有 2- 发射超标量——可以在每个时钟周期最多发射 2 条指令——拥有以下特性：

- 编译器必须将指令组织成最多 2 段的 ILP。
- 如果需要，则指令流水线包含足够的资源（加法器 / 减法器、乘法器、除法器）来并行执行两条算术指令。为了简单起见，假定每条算术计算只需要 1 个时钟周期。
- 调度“LD”指令尽早执行从而隐藏从 DM 中加载数据所需要的一个周期的延迟。

例 8-14 中列出了编译器对示例 8-13 中指令采用静态调度处理从而生成的 2- 发射指令组织结构。表 8-8 说明了程序代码的执行，其中假定变量 a 和变量 b 已经加载到 cache 中。在每个流水线周期中，两条指令并行进行取指和执行操作。

表 8-8 使用 5 级超标量流水线执行静态调度的 2- 发射的程序代码

| 周 期 | 取指令 | 译 码 | 执 行 | 数据存储器 | 写 回 |
|-----|------------------------------------|------------------------------------|------------------------------------|------------------------------------|------------------------------------|
| 1 | 1: ADD R3, R1, R2 5: LD R5, (d) | | | | |
| 2 | 2: MUL R4, R3, R3 NOP | 1: ADD R3, R1, R2 5: LD R5, (d) | | | |
| 3 | 3: ST (c), R4 4: SUB R3, R2, R1 | 2: MUL R4, R3, R3 NOP | 1: ADD R3, R1, R2 5: LD R5, (d) | | |
| 4 | 6: ADD R4, R3, R5 NOP | 3: ST (c), R4 4: SUB R3, R2, R1 | 2: MUL R4, R3, R3 NOP | 1: ADD R3, R1, R2 5: LD R5, (d) | |
| 5 | 7: DIV R6, R4, R1 NOP | 6: ADD R4, R3, R5 NOP | 3: ST (c), R4 4: SUB R3, R2, R1 | 2: MUL R4, R3, R3 NOP | 1: ADD R3, R1, R2 5: LD R5, (d) |
| 6 | | 7: DIV R6, R4, R1 NOP | 6: ADD R4, R3, R5 NOP | 3: ST (c), R4 4: SUB R3, R2, R1 | 2: MUL R4, R3, R3 NOP |
| 7 | | | 7: DIV R6, R4, R1 NOP | 6: ADD R4, R3, R5 NOP | 3: ST (c), R4 4: SUB R3, R2, R1 |
| 8 | | | | 7: DIV R6, R4, R1 NOP | 6: ADD R4, R3, R5 NOP |
| 9 | | | | | 7: DIV R6, R4, R1 NOP |

例 8-14 对例 8-13 中指令采用静态调度使之可执行 2- 发射 ILP，流水线可并行执行两条算术指令。

ADD R3, R1, R2
MUL R4, R3, R3
ST (a), R4
ADD R4, R3, R5
DIV R6, R4, R1

LD R5, (b)
NOP
SUB R3, R2, R1
NOP
NOP



因为一个超标量处理器可在每个时钟周期内执行多条指令，所以程序的 CPI 通常小于 1。因为这个原因，流水线首选的性能参数为单位周期内指令数（IPC），IPC 通过对 CPI 求倒数得到，如公式（8-6）所示。

$$IPC = \frac{\text{指令执行的数量 } (n)}{\text{使用的时钟周期数量 } (N)}$$

(8-6)

355

若不计入填充流水线的时钟周期数，例 8-14 中的 IPC 如表 8-8 所示，计算如下：

$$N = 9 - 5 + 1 = 5 \text{ 个时钟周期 (忽略填充流水线的开始部分)}$$
$$n = 7 \text{ 条执行指令 (例 8-13 中的程序)}$$
$$IPC = \frac{7}{5} = 1.4$$

(8-7)

一种进一步提升 IPC 的技术叫作基于编译器的随机执行。在这种情况下，分支方向独立的指令将被选择并行执行。在英特尔安腾架构下，采用基于编译器的随机执行方式，将一个“if-else”指令语句转换为一个称为已预测的指令的条件指令。例如，考虑下面简单的“if-else”语句：

```
if(a > 0)
    a = a - 1;
else
    a = a + 1;
```

编译器将转换 if-else 语句为下列的汇编代码，其中编译器将把条件“a > 0”附给所有“then”代码部分的指令（在这个示例中），而把条件“!(a > 0)”附给所有的“else”代码部分，如下所示。这将消除其他与“then”或“else”代码部分相关的分支指令。

```
LD R1, (a)
CMP R1, 0
SUB R1, R1, 1 (GTF = 1)    //if greater than flag (GTF) is set
ADD R1, R1, 1 (GTF = 0)    //if GTF is not set
ST (a), R1
```

“SUB”指令和“ADD”指令都可以通过调度并行执行。处理器将计算 a - 1 和 a + 1。然而，这些结果只会有一个写入寄存器 R1 中，具体为哪一个由 GTF 标志寄存器的值来确定。如果“a > 0”，R1 将采用 a - 1 的结果，否则，R1 将采用 a + 1 的结果。因为某个计算结果可能无法使用，所以这种计算（a - 1 或 a + 1）是随机的。

采用静态调度的 ILP 的一大优势是其处理器能耗更高，因为其在每个时钟周期中不使用硬件来决定对某个指令集并行执行。因为这个原因，在手持设备（如智能手机）中常使用采用静态调度的 ILP 处理器（如 ARM Cortex-A8）。与采用动态调度的 ILP 处理器（下面讨论）相比，采用静态调度的 ILP 处理器的另一优势是编译器可以为 ILP 检查更长的指令列表，而在动态调度的 ILP 中，在每个时钟周期内必须动态且快速地检测指令列表。

2. 动态调度的 ILP

相对于采用静态调度的 ILP，在超标量处理机中采用动态调度的 ILP 的流水线复杂度更高、功率更高。在流水线取指令阶段含有一个存储尚未进行调度处理指令的队列。使用专用硬件动态检测队列中的多条指令，并决定哪些指令可并行执行。与采用静态调度的 ILP 的流水线相比，采用动态调度的 ILP 的流水线具有以下优势：

- 确定的程序流只会在运行时才能知道。因此，对于处理器而言，有更多的机会来选择和发射多条指令并行执行。

- 处理器也可以实现寄存器重命名机制，其使用程序员不可见的临时寄存器代替指令中某些寄存器的名字，从而动态地消除程序中指令间的反相相关性和输出相关性。这允许处理器调整指令顺序来实现更高的指令级并行。
- 当执行程序的处理器更新为下一代采用动态调度的 ILP 的处理器或其动态调度实现方式有所不同时，程序并不需要重新编译来获取采用动态调度的优势。

例如，Intel 酷睿 i7 处理器的流水线采用了动态调度的 ILP 技术。其较高的指令吞吐量使得其适用于高端台式机或服务器。

动态指令调度要求流水线采用诸如计分牌或更高级版本的被称为 Tomasulo's 算法等特定的技术，这些技术也支持随机执行。流水线可动态地给寄存器重命名从而避免 WAR 和 WAW 冒险，并允许改变指令执行顺序。例如，考虑在之前例 8-13 中讨论的包含潜在的 5 个 RAW、2 个 WAW 和 1 个 WAR 冒险的程序代码（图 8-26）。

图 8-27 说明了如何通过重命名技术来消除程序中的两个反相相关性和一个输出相关性。通过使用临时寄存器 T1 重命名“ADD”指令中的目的寄存器 R3，消除了指令“ADD R3, R1, R2”和指令“SUB R3, R2, R1”之间的输出相关性。这是因为，在程序执行中，使用“ADD”指令执行结果的指令“MUL R4, R3, R3”位于需要“SUB”指令执行结果的指令“ADD R4, R3, R5”之前。“MUL”指令中的 R3 同样使用 T1 重命名。这样修改同时也消除了“MUL”和“SUB”指令之间的反相相关性。

与之类似，通过使用临时寄存器 T2 重命名“MUL”指令中的 R4 寄存器，从而消除在“MUL”和“SUB”指令之间的输出相关性。指令“ST (a), R4”的 R4 同样使用 T2 重命名。这样指令“ADD R4, R3, R5”中的 R4 没有改变，当程序每次执行一条指令时，这条指令的执行结果，将像它应该的那样，存入 R4 寄存器中。

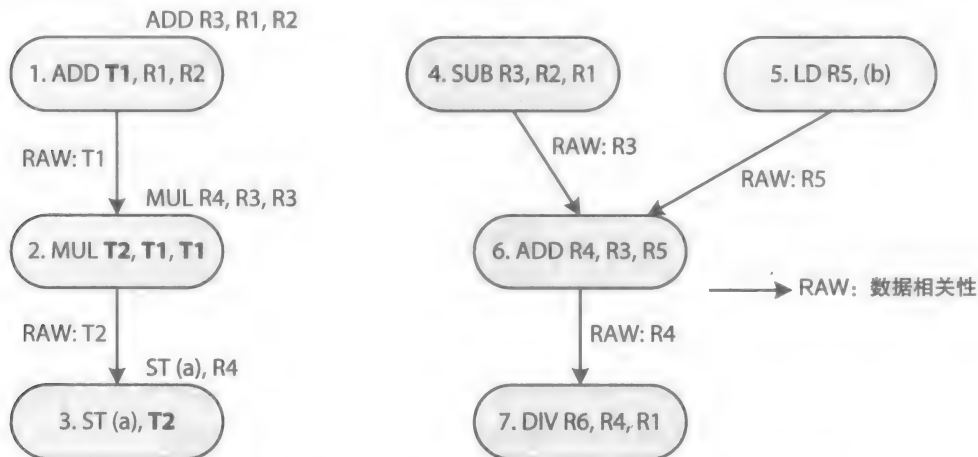


图 8-27 使用重命名技术消除图 8-26 中 WAR 和 WAW 相关性

表 8-9 程序代码采用 2-发射动态调度的执行过程

| 周 期 | 取指令 | 译 码 | 执 行 | 数据存储器 | 写 回 |
|-----|--|------------------------------------|-----|-------|-----|
| 1 | 1: ADD T1, R1, R2 5: LD R5, (b) | | | | |
| 2 | 2: MUL T2, T1, T1 4: SUB R3, R2, R1 | 1: ADD T1, R1, R2 5: LD R5, (b) | | | |

(续)

| 周 期 | 取指令 | 译 码 | 执 行 | 数据存储器 | 写 回 |
|-----|-----------------------------------|--|--|--|--|
| 3 | 3: ST(a), T2 6: ADD R4, R3, R5 | 2: MUL T2, T1, T1 4: SUB R3, R2, R1 | 1: ADD T1, R1, R2 5: LD R5, (b) | | |
| 4 | 7: DIV R6, R4, R1 Nop | 3: ST(a), T2 6: ADD R4, R3, R5 | 2: MUL T2, T1, T1 4: SUB R3, R2, R1 | 1: ADD T1, R1, R2 5: LD R5, (b) | |
| 5 | | 7: DIV R6, R4, R1 Nop | 3: ST(a), T2 6: ADD R4, R3, R5 | 2: MUL T2, T1, T1 4: SUB R3, R2, R1 | 1: ADD T1, R1, R2 5: LD R5, (b) |
| 6 | | | 7: DIV R6, R4, R1 Nop | 3: ST(a), T2 6: ADD R4, R3, R5 | 2: MUL T2, T1, T1 4: SUB R3, R2, R1 |
| 7 | | | | 7: DIV R6, R4, R1 Nop | 3: ST(a), T2 6: ADD R4, R3, R5 |
| 8 | | | | | 7: DIV R6, R4, R1 Nop |

表 8-9 说明了在一个采用 2- 发射动态调度的超标量处理器上使用图 8-27 中修改后的指令的程序执行过程。对程序也有下列要求，这与在采用 2- 发射静态调度的超标量处理器上的要求相同。

- 如果需要，则指令流水线包含足够的资源（加法器 / 减法器、乘法器、除法器）来并行执行两条算术指令。为了简单起见，假定每条算术计算只需要 1 个时钟周期。
- 调度“LD”指令尽早执行从而隐藏从 cache 中加载数据所需要的一个周期的延迟。

在第一个流水线周期中，3 条独立的指令“ADD T1, R1, R2”、“SUB R3, R2, R1”和“LD R5, (b)”，如图 8-27 所示，至少有 2 条必须发射执行。因为“ADD”指令在“SUB”指令之前，故而“ADD”指令被选中。“LD”指令被选中，从而可以更早地加载其中的数据，从而隐藏“LD”指令在指令“ADD R4, R3, R5”之前执行的必要的一个时钟周期的延迟。

在第 2 和第 3 个周期，调度部件将发射两个独立的指令“MUL T2, T1, T1”和“SUB R3, R2, R1”，之后再发射另两条独立的指令“ST (a), T2”和“ADD R4, R3, R5”。最后，在第 4 个周期发射指令“DIV R6, R4, R1”。正常情况下，当前指令调度发射后，新的指令将取值并存储到这一指令队列中。然而，为了使配图简单化，这样的步骤在此处省略。

359

假定忽略最初的填充流水线所需的时钟周期，程序代码的 IPC 由公式（8-8）决定，为 1.75。在这个程序示例中，使用动态调度的 2- 发射超标量处理器比同等的采用静态调度的超标量处理器快 25%（1.75/1.4 = 1.25）。

$$\begin{aligned} \text{IPC} &= \frac{7 \text{ 条指令}}{(8 - 5 + 1) \text{ 个周期}} \\ &= \frac{7 \text{ 条指令}}{4 \text{ 个周期}} = 1.75 \end{aligned}$$

(8-8)

然而，因为一个典型的 RISC 处理器有许多（例如，32 个）寄存器，为避免使用最近使用的寄存器，编译器可以给指令分配不同的寄存器，从而减少指令之间的一些反相相关性和输出相关性，从而实现更好的静态调度的 ILP。

采用动态调度时也可能采用随机执行。在这种情况下，一旦指令的操作数可用，并且

与分支决定无关，处理器将执行指令。如果程序流发生改变，一些随机的计算结果可能会被丢弃，当每个时钟执行一条指令时，需要提交给寄存器或存储器的计算结果必须按程序顺序完成。

8.4.4 多线程

正如我们在第1章和本章中早先提到的那样，ILP 存在一个限制。对于一个给定的程序，即使采用寄存器重命名技术，使具有相关性的指令可在每个流水线周期中同时执行，程序中独立的指令依然很少。一些基准测试程序的研究显示，约30%的时间3条指令可并行执行，约2%的时间6条或多条指令可并行执行，平均每个时钟周期可并行执行2.5条指令[11]。即使不限制可用晶体管的数量和设备最大功耗，一个程序执行的时间也有限制。因此，更快执行一个任务的唯一方式是分割这个任务为多个可同时执行的子任务。

1. 程序示例

考虑下面给出的C语言程序，其中有一个一千万次迭代的for循环。为了简单起见，数组的元素全部初始化为1.0。

```
main()
{
    double array[100000000] = {[0 ... 99999999] = 1.0};
    double sum;
    int i;
    sum = 0.0;
    for(i = 0; i < 100000000; i++)
        sum = sum + array[i];
    printf("sum = %e\n", sum);
}
```

360

降低计算数组中各元素之和所需要的时间的一种方法是将数组分割为2等份，使用两个不同的线程分别计算每个等份中各个元素之和。例如，下面使用C语言编写的双线程程序可用来计算数组中各个元素之和。程序中，“main”函数首先创建一个线程（线程0）来求数组中前半部分各元素之和，之后“main”函数本身，作为线程1，计算数组中剩余部分各元素之和。一旦“main”函数（现在为线程1）完成后半部分元素求和，将检查确定线程0是否完成前半部分元素求和工作。若线程0完成，“main”函数将计算并打印两部分元素之和。注意，两个线程都可以访问声明为全局变量的“array”和“sum”。

```
/*compile as "gcc -pthread filename.c" */
#include <pthread.h>
#include <stdio.h>

//globally declared array and sum
double array[100000000] = {[0 ... 99999999] = 1.0};
double sum[2]; //to store two partial sums

typedef struct {
    int start;
    int end;
    int pid;
} range;

void *calculate_array_sum(void *arg)
{
    range *incoming = (range *) arg;
    int i;
```



```

double partial_sum;
int start, end, pid;

start = incoming->start;
end = incoming->end;
pid = incoming -> pid;

partial_sum = 0;
for(i = start; i < end; i++)
    partial_sum = partial_sum + array[i];

sum[pid] = partial_sum;

return;
}

main ()
{
    pthread_t threadID;
    void *exit_status;
    range range_thread0;
    range range_thread1;

    //define the array range for Thread 0
    range_thread0.start = 0;
    range_thread0.end = 50000000;
    range_thread0.pid = 0; //Thread 0

    //now create a thread to sum the first half of the array elements
    pthread_create(&threadID, NULL, calculate_array_sum,
    &range_thread0);

    //define the array range for the main as Thread 1
    range_thread1.start = 50000000;
    range_thread1.end = 100000000;
    range_thread1.pid = 1; //Thread 1
    calculate_array_sum(&range_thread1);

    pthread_join(threadID, &exit_status); //wait for Thread 0

    printf("Grand total = %e\n", sum[0] + sum[1]);
}

```

361

上面 C 语言程序的执行将产生两部分代码，如下所示，当 CPU 采用多线程时，两部分代码可并行执行。

| 线程 0 | “main” 作为线程 1 |
|-------------------------------|---------------------------------------|
| for(i = 0; i < 50000000; i++) | for(i = 50000000; i < 100000000; i++) |
| partial_sum = partial_ | partial_sum = partial_ |
| sum + array[i]; | sum + array[i]; |
| sum[0] = partial_sum; | sum[1] = partial_sum; |

指令流水线必须包含所有寄存器（包括 PP）的两份副本。下面章节将描述 3 种 k -发射多线程的流水线组织结构。

2. 粗粒度

每次执行 LD 指令或 ST 指令，都将造成一个长等待，例如当需要访问主存时，流水线将切换并唤醒另一个进程，使之开始执行。例如，当执行上述的线程 0 和线程 1 时，流水线在线程 0 中连续发射 k 条指令，直到其访存的数据不在 cache 中而必须从主存储器中访问得到，这将有一段很长的延迟。此时，当线程 0 从主存中访问数据，流水线将切换到线程 1 并连续发射 k 条指令，直到线程 1 遇到一个长延迟时，流水线再切换回线程 0 并发射 k 条指令。

362 这被称为一个粗粒度的多线程体系结构，因为流水线在一个时间内执行一个线程，直到它遇到一个很长的延迟时才进行切换。因为每个线程使用其私有的寄存器集，被切换的线程将自动保存其状态。这种方式在延迟很长的情况下效果很好，但当延迟较短时，效果并不显著，例如，当从一个低等级的 cache 中访问数据时，若访问延迟要比上述小很多时，其效果将比较差。这是因为每次切换线程时，流水线必须被清空，这将浪费一定的流水线周期。此外，在某些应用程序中，线程切换发生的频率可能比必需的还少。例如，在一些实时应用中需要线程更快执行。

流水线组织结构与多线程结构相比，有一大优点，即其所需要的硬件数量要比多线程少，这是因为流水线中的资源在任一时间都只用于一个线程的执行。另一个优点是，它可以用于实现一个采用静态或动态 ILP 的多线程处理器。注意，总体而言，线程可以属于也可以不属于同一程序。两个不同程序的线程也可以同时执行。

3. 细粒度

在这种情况下，线程每个时钟周期都进行切换。例如，当执行上述的线程 0 和线程 1 时，流水线在第 1 个流水线周期在线程 0 中最多发射 k 条指令，之后流水线将进行切换，在第 2 个流水线周期在线程 1 中最多发射 k 条指令，依次重复进行。同样，一般情况下，线程可以属于也可以不属于同一个程序。

这被称为细粒度的多线程体系结构，因为流水线在每一个周期都进行了线程切换。与粗粒度的多线程流水线相比，该流水线效率更高，但是每个线程的执行速度仍然很慢。当流水线在每一个周期中只执行一个线程的指令时，其他线程必须等待。例如，在 Sun 的 Niagara 处理器和 Nvidia GPU 上，实现这种类型的多线程。

因为线程在每个时钟周期进行交换，与粗粒度多线程结构相比，细粒度多线程结构提供了一个更高的线程级并发。与之类似，这一结构也可以用于实现一个采用静态或动态 ILP 的多线程处理器。

4. 同步

363 在这种情况下，流水线在每个时钟周期从所有运行的线程中最多选取 k 条指令进行发射。例如，当执行上述的线程 0 和线程 1 时，流水线在每个时钟周期从线程 0 中发射 k_0 ($0 \leq k_0 \leq k$) 条指令，从线程 1 中发射 k_1 ($0 \leq k_1 \leq k$) 条指令，其中 ($0 \leq k_0 + k_1 \leq k$)。因为流水线从多个线程中选择独立不相关的指令，并同步执行，这种结构也被叫作同步多线程。这种结构的流水线必须实行动态调度的 ILP，因此，同步多线程的实现要求最多数量的硬件资源。相比对于之前的多线程结构，在线程级并行 (TLP) 方面，同步多线程结构的优

势在于可同时执行来自多个线程的多条指令。

例 8-15 考虑下面两种程序代码的执行，标记为线程 A 和线程 B，其在采用了同步多线程、动态调度 ILP 的 4-发射超标量处理器上执行。线程 A 执行例 8-13 的程序代码，其指令间包含数据相关性、反相相关性以及输出相关性。为了简单起见，线程 B 中的指令之间只包含数据相关性。

线程 A (例 8-13 中的程序代码)

```
ADD R3, R1, R2
MUL R4, R3, R3
ST (a), R4
SUB R3, R2, R1
LD R5, (b)
ADD R4, R3, R5
DIV R6, R4, R1
```

线程 B

```
ST (x), R3
ADD R4, R1, R2
MUL R6, R4, R5
DIV R7, R3, R6
ST (z), R7
LD R8, (y)
SUB R9, R7, R8
```

假设以下：

- 如果需要，执行阶段有足够的资源（例如，加法器 / 减法器、乘法器、除法器），可并行执行任何 4 个算术指令。为简单起见，也假定每个算术运算都需要一个时钟周期。
- DM（cache）组织结构采用 2- 路交叉存储（第 7 章），流水线采用可同步执行 LD 和 ST 指令的设计。假定在一个采用交叉存储的存储器中同步执行一个 ST 指令和一个 LD 指令比同步执行两个 ST 指令和两个 LD 指令遇到冲突的可能性小。
- 流水线将尽可能早地发射 LD 指令从而隐藏 LD 指令从 cache 中加载数据所必需的一个周期的延迟。
- 变量 a、b 和 x ~ z 存储在 cache 中，没有冲突。

表 8-10 说明了例 8-15 中线程 A 和线程 B 的执行过程。在流水线周期 1 中，流水线从线程 A 中发射两条指令“ADD T1,R1, R2”和“LD R5, (b)”，从线程 B 中发射另两条指令“ST (x), R3”和“ADD R4, R1, R2”。在流水线周期 2 中，流水线继续从线程 A 中发射两条指令，从线程 B 中发射另两条指令。在流水线周期 3 中，流水线从线程 A 发射两条指令，从流水线 B 发射 1 条指令。最后，在流水线周期 4 中，流水线最后发射两条指令，各从每个线程中发射一条指令。每个周期中发射的指令是数据不相关的，可并行执行。因为流水线在每个周期只能发射一条 LD 指令和一条 ST 指令，指令“LD R8, (y)”不能在周期 1 中更早发射。因此，指令“SUB R9, R6, R8”的执行延迟到第 6 个周期执行。

364

表 8-10 使用双线程在采用同步多线程、动态调度的 ILP 的 4- 发射超标量处理器上程序代码的执行过程

| 周 期 | 取指令 | 指令译码 | 指令执行 | 数据存储器 | 数据写回 |
|-----|--|--|--|--|--|
| 1 | A: ADD T1, R1, R2 A: LD R5, (b) B: ST (x), R3 B: ADD R4, R1, R2 | | | | |
| 2 | A: MUL T2, T1, T1 A: SUB R3, R2, R1 B: MUL R6, R4, R5 B: LD R8, (y) | A: ADD T1, R1, R2 A: LD R5, (b) B: ST (x), R3 B: ADD R4, R1, R2 | | | |
| 3 | A: ST (a), T2 A: ADD R4, R3, R5 B: DIV R7, R3, R6 B: SUB R9, R6, R8 | A: MUL T2, T1, T1 A: SUB R3, R2, R1 B: MUL R6, R4, R5 B: LD R8, (y) | A: ADD T1, R1, R2 A: LD R5, (b) B: ST (x), R3 B: ADD R4, R1, R2 | | |
| 4 | A: DIV R6, R4, R1 B: ST (z), R6 ? ? | A: ST (a), T2 A: ADD R4, R3, R5 B: DIV R7, R3, R6 B: SUB R9, R6, R8 | A: MUL T2, T1, T1 A: SUB R3, R2, R1 B: MUL R6, R4, R5 B: LD R8, (y) | A: ADD T1, R1, R2 A: LD R5, (b) B: ST (x), R3 B: ADD R4, R1, R2 | |
| 5% | | A: DIV R6, R4, R1 B: ST (z), R6 ? ? | A: ST (a), T2 A: ADD R4, R3, R5 B: DIV R7, R3, R6 O | A: MUL T2, T1, T1 A: SUB R3, R2, R1 B: MUL R6, R4, R5 B: LD R8, (y) | A: ADD T1, R1, R2 A: LD R5, (b) B: ST (x), R3 B: ADD R4, R1, R2 |
| 6 | | | A: DIV R6, R4, R1 B: ST (z), R6 B: SUB R9, R6, R8 ? | A: ST (a), T2 A: ADD R4, R3, R5 B: DIV R7, R3, R6 O | A: MUL T2, T1, T1 A: SUB R3, R2, R1 B: MUL R6, R4, R5 B: LD R8, (y) |

(续)

| 周 期 | 取指令 | 指令译码 | 指令执行 | 数据存储器 | 数据写回 |
|-----|-----|------|------|--|--|
| 7 | | | | A: DIV R6, R4, R1 B: ST (z), R6 B: SUB R9, R6, R8 ? | A: ST (a), T2 A: ADD R4, R3, R5 B: DIV R7, R3, R6 O |
| 8 | | | | | A: DIV R6, R4, R1 B: ST (z), R6 B: SUB R9, R6, R8 ? |

% 由于 “LD R8, (y)”, “SUB R9, R6, R8” 延迟 1 个时钟周期。

A: 线程 A 指令。

B: 线程 B 指令。

基于 4- 发射的 ILP 如表 8-10 所示，对线程 A 或线程 B 的 IPC 均为 1.75 (7/4)，其中忽略填充流水线所需要的时钟周期。然而，采用同步多线程的线程 A 和线程 B 的 IPC——同样也忽略填充流水线所需要的时钟周期——为 3.5，计算方式见公式 (8-9)。

365

$$\begin{aligned} \text{IPC} &= \frac{(7 + 7) \text{ 条指令}}{(8 - 5 + 1) \text{ 个周期}} \\ &= \frac{14 \text{ 条指令}}{4 \text{ 个周期}} = 1.75 \end{aligned}$$

(8-9)

多线程，尤其是同步多线程结构，提升了流水线的效率。此外，同步多线程流水线实现小规模 TLP。更高级的 TLP 要求采用多核处理器或多处理器系统，这将在第 10 章中进行讨论。

参考文献

1. IEEE standard for microprocessor assembly language (IEEE Std. 694-1985), IEEE, 1985.
2. MASM, <http://www.masm32.com/>.
3. cygWin (GNU + Cygnus + Windows), <http://www.cygwin.com/>.
4. Intel Architecture Optimization Manual, 1997, www.intel.com.
5. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2014.
6. Simics (system level instruction set simulator), <http://www.virtutech.com/>
7. J. E. Smith, A study of branch prediction strategies, *Proceedings of the 8th Annual International Symposium on Computer Architecture*, June 1981, pp. 135-147.
8. Shien-Tai Pan, Kimrning So, Joseph T. Rahmeh, Improving the accuracy of dynamic branch prediction. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Sept 1992, pp. 76-84.
9. M.-C. Chang and Y.-W. Chou, Branch prediction using both global and local branch history information, *IEE Proc-Comput Digit Tech*, Vol. 149, No. 2, 2002, 33-38.
10. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, Waltham, 2012.
11. David Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel Computer Architecture: Hardware/Software Approach*, Morgan Kaufmann, San Francisco, 1999.

练习

8.1 思考一个采用累加器 ISA 的 CPU 执行下列的伪代码，完成以下任务：

```
... //some code
while{(true) //loop forever
{
    T = (-5 - A + B) ^ C; //where ^ stands for bit-wise XOR
```

```

... //some other code (not given)
}

```

- a. 设计一个指令长为 8 位的累加器 ISA 指令集，其中操作码 3 位，操作数 5 位。A ~ C 中的任一变量可能为负的 2 的补码。
 - b. 使用刚设计的指令集编写汇编程序，假定代码从存储器地址 0 开始，每次递增 1，数据从存储器地址 0x1F 开始，每次递减 1。
 - c. 手动汇编你的汇编程序，使用二进制和十六进制编写你的程序。从地址 0 开始顺序分配操作码给相应的汇编程序指令。
 - d. 绘制 CPU 数据通路图，只需画出数据通路，其他部件可省略。
- 8.2 思考累加器 ISA 的汇编指令 “LD data”(ACC \leftarrow data), “LD (adrs)”(ACC \leftarrow Memory[adrs]), “ST (adrs)”(Memory[adrs] \leftarrow ACC), “ADD (adrs)”(ACC \leftarrow ACC + Memory[adrs]), “XOR (adrs)”(ACC \leftarrow ACC \oplus M[adrs])。完成以下任务：

366

- a. 为下面程序编写汇编程序：


```

X = -2;
Y = 6;
Z = 11;
T = X + Y - Z;

```
 - b. 针对上述汇编指令，绘制单周期指令数据通路。
- 8.3 累加器 ISA CPU 执行下列指令，使用 3 位的操作码和 5 位的地址或 2 的补码数。完成以下任务：

```

LD (address)    //Acc←Memory [address], read from LM2
LD data         //Acc←data (a 2's complement number, sign
                //extended)
ADD data        //Acc←Acc + data (data is a 2's complement
                //number, sign extended)
SUB data        //Acc←Acc - data (data is a 2's complement
                //number, sign extended)
ADD (address)   //Acc←Acc + Memory[address]
SUB (address)   //Acc←Acc - Memory[address]
ST (address)    //M[address]←Acc
JMP address     //PP←address
JZ address      //PP←address if ACC = 0

```

- a. 绘制 CPU 的数据通路，假定 DM 有独立分开的输入和输出总线，如图 8-7 所示，不用包含未被上述指令使用的数据通路。
 - b. 绘制 CPU 的数据通路，假定 DM 有一个双向数据总线。不用包含未被上述指令使用的数据通路。
- 8.4 对下列显示的更高难度的代码段，创建一组指令，分别为下列体系结构编写一个等价的汇编程序：

```

int i, sum;
for (i=0; i < k, i++)
    if (i mod 2 == 0)
        sum = sum + i;

```

- a. Stack-ISA
 - b. Acc-ISA
 - c. CISC-ISA
 - d. RISC-ISA
- 8.5 使用寄存器设计一个简单的 8 位宽、深度为 16 的硬件堆栈。其包含 4 个信号线，为 push、pop、

367

ovf (栈顶溢出) 和 udf (栈底溢出)。假定信号高电平有效。当栈为空时, 进行 push 操作, ovf 将变为 1 (选中), 当栈为空时, 进行 pop 操作, udf 将为 1。

- 8.6 讨论可变的 CISC 指令的格式与固定的 RISC 指令格式是如何使得指令数据通路的设计变得复杂。
- 8.7 给定图 8-7 中的单周期 CPU 数据通路, 请估计时钟频率的最大值。假定 Δ_{IM} 和 Δ_{DM} 均为 1.2ns, Δ_{add} 为 0.8ns, $\Delta_{add/cmp}$ 为 0.9ns, Δ_{mux} 为 0.3ns, Δ_{NAND} 为 0.1ns, τ_{st} 、 τ_{cq} 、 τ_{cs} 均为 0.05ns。
- 8.8 给定图 8-11 中的流水线数据通路, 请估计时钟频率的最大值, 假定 Δ_{IM} 和 Δ_{DM} 均为 1.2ns, Δ_{add} 为 0.8ns, $\Delta_{add/cmp}$ 为 0.9ns, Δ_{2-1MUX} 为 0.3ns, Δ_{NAND} 为 0.1ns, τ_{st} 、 τ_{cq} 、 τ_{cs} 均为 0.05ns。
- 8.9 思考包含取指令 (F)、译码 (D)、执行 (E) 和写回 (WB) 4 个阶段的流水线, 其中 Δ_E 为其他阶段的两倍。当流水线执行 n 条指令时完成以下内容:
- 假设 E 阶段可被分为两个阶段 E1 和 E2, 其中 $\Delta_{E1} = \Delta_{E2} = 1/2 \Delta_E$ 。假定 CPI 为 1, n 趋近于无穷时, 根据 Δ_E 和 $\Delta_{clocking}$ 确定加速比的表达式。此外, 请估算当 n 趋近于无穷且 $\Delta_E = 2ns$ 和 $\Delta_{clocking} = 0.1ns$ 时的加速比。
 - 假设 E 阶段可使用该阶段的 2 个硬件和 MUX 的副本进行超标量。假定 CPI 为 1, n 趋近于无穷时, 根据 Δ_E 、 $\Delta_{clocking}$ 和 Δ_{MUX} 确定加速比的表达式。此外, 请估算当 $\Delta_E = 2ns$ 、 $\Delta_{clocking} = 0.1ns$ 、 $\Delta_{MUX} = 0.3ns$ 情况下, n 趋近于无穷时的加速比。
- 8.10 讨论交换图 8-17 中 E 阶段和 DM 阶段位置的效果。
- 8.11 根据给定的例 8-10 中的汇编代码, 完成以下工作:
- 根据图 8-17 中的 5 级流水线绘制 2 次 for 循环迭代的流水线图, 不用包含 for 循环之后的指令 (如 Ix、Iy、Iz) 的执行。
 - 计算两次迭代时程序的 CPI。
 - 确定求 k 次迭代时 CPI 的计算公式。
 - 确定 k 趋向于无穷时 CPI 的极限值。
- 8.12 根据给定的例 8-11 中的汇编代码, 完成以下工作:
- 根据图 8-17 中的 5 级流水线绘制 2 次 for 循环迭代的流水线图, 不用包含 for 循环之后的指令 (如 Ix、Iy、Iz) 的执行。
 - 计算两次迭代时程序的 CPI。
 - 确定求 k 次迭代时 CPI 的计算公式。
 - 确定 k 趋向于无穷时 CPI 的极限值。
- 8.13 根据表 8-6 中的流水线图, 确定迭代次数 k 为 10 时 CPI 的计算公式。并确定当 k 趋向于无穷时 CPI 的下限值。
- 8.14 考虑下列包含一个单独的 if-else 语句的 for 循环以及其编译器生成的分支指令。假定处理器采用基于分支相关性的预测器。在 for 循环执行过程中, 当 i 为偶数 (如 0、2、4 等) 时, 将执行 “then” 部分代码, 当 i 为奇数时, 将执行 “else” 部分代码。完成以下工作:

368

| | |
|--|---|
| <pre> for (i = 0; ...) { if (... > ...) ... else ... } </pre> | <pre> L1: ... CMP ... JGT endfor CMP ... JLE else ... JMP endif else: endif: JMP L1 endfor: ... </pre> |
|--|---|

a. 假设预测器采用 2 位的 BHR。填写下列进行 6 次迭代的表格，并确定错误预测的数量，其中“N”(预测不跳转)、“T”(预测跳转)、“LN”(预测可能不跳转)、“LT”(预测可能跳转)是用于说明 2 位预测器的状态，每一个状态对应 BHR (NN、NY、YN、YY) 的一条执行路径。在表中，前一行中的状态被用来预测当前指令的分支方向为 Y 或 N。例如，开始部分，当执行“JMP endif”指令时，其中 BHR = NN (“BGT”指令不跳转，“BLE”指令不跳转)，因为“JMP endif”发生跳转，BHR 为 NN 的 2 位预测被初始化为“T”。当 BHR 下一次为 NN 时，相关预测器将为当前指令预测“跳转”。BHR = NY 如表中所示。

| | | 分支指令 | | 2 位预测器 | | | |
|--------|-----------|-----------|-----|-------------|-----|------|----------|
| 迭 代 | 前一指令 | 当前指令 | BHT | 表 BHT 初始化状态 | 预 测 | | 下一状态 |
| 1 | JGT | JLE | NN | T | | | “then” |
| | BLE | JMP endif | NY | T | | | |
| | JMP endif | JMP L1 | YY | N | | | |
| | IMP L1 | JGT | YN | T | | | |
| 2 | JGT | JLE | NY | | 跳转 | | T “else” |
| | JLE | JMP L1 | YY | | 不跳转 | | N |
| | JMP L1 | JGT | YN | | 跳转 | Miss | LT |
| | | | | | | | |

b. 假定预测器采用 3 位的 BHR 来确定包括 NNN、NNT、NTN、NTT、TNN、TNT、TTN 和 TTT 共 8 条执行路径中的一条。然而，具体执行哪一条路径由程序执行时处理的数据决定。

369

| | | 分支指令 | | | | 2 位预测器 | | | |
|-----|---|-----------|-----------|-----------|-----|-----------------|--------|------|--------|
| 迭 | 代 | 前两条指令 | | 当前指令 | BHR | 表 BHT 初 始化状态 | 预 测 | 下一状态 | |
| 1 | | JGT | JLE | JMP endif | NNY | T | | | “then” |
| | | JLE | JMP endif | JMP L1 | NYY | N | | | |
| 2 | | JMP endif | JMP L1 | JGT | NYN | T | | | “else” |
| | | JMP L1 | JGT | JLE | YNY | T | | | |
| | | JGT | JLE | JMP L1 | NYY | | 不跳转 | N | |
| ... | | | | | | | | | |

8.15 思考下面的程序代码，假定 1) DM 采用交叉存储方式，任何两条存储指令可同时执行，2) 每条算术指令需要 1 个时钟周期来执行。完成以下工作：

```
LD R1, (a)
LD R2, (b)
ADD R3, R1, R2
ST (c), R3
LD R4, (d)
MUL R5, R3, R4
ST (d), R5
SUB R3, R2, R1
DIV R6, R3, R5
ST (e), R6
```

- a. 调整指令执行顺序，使之能能在一个采用静态调度的 2- 发射超标量处理器上执行。
- b. 调整指令执行顺序，使之能能在一个采用动态调度的 2- 发射超标量处理器上执行。

8.16 思考例 8-13 中的程序代码。假定采用动态调度的 2- 发射 (ILP) 超标量处理器调度算术指令而

不是“LD”指令尽早执行。使用一个流水线图说明程序执行过程，并计算 IPC。当计算 IPC 时忽略填充流水线所需要的时钟周期。

- 8.17 思考一个 4- 发射同步多线程的超标量处理器。并思考一个多线程的多线程程序，其中每个线程执行约 10^{11} 条指令。完成以下工作：
- a. 当没有存储器访问延时，假定程序的 IPC 为 3.5。假设时钟频率为 1GHz，处理器执行这个程序需要多长时间？忽略因为操作系统过载造成的延时。
 - b. 假定程序中 LD 指令和 ST 指令占 20%。这些指令中 10% 将导致数据加载和写回的存储器延时，使得 IPC 降低为 1.75。请确定程序执行的时间。

计算机安全

- 8.18 计算机安全（安全的协处理器）：练习 11.27（也可参见 11.4 节、11.8 节和 11.10 节）。
- 8.19 计算机安全（安全的处理器）：选择练习 11.28 和 / 或练习 11.29（也可参见 11.4 节 11.9.2 节和 11.11 节）。
- 8.20 计算机安全（欺骗、拼接和重放攻击）：练习 11.30（也可参见 11.3 节和 11.11 节）。
- 8.21 计算机安全（安全处理器性能相关问题）：练习 11.31（也可参见 11.11 节）。

370
2
371

计算机体系结构：互连

9.1 简介

现代计算机系统是由一个或多个处理器、存储器单元和输入/输出(I/O)设备构成的互连系统。个人计算机(如微型计算机)还可以包括一个可选的专用或自定义处理器用作加速器(如GPU、FPGA)。键盘、鼠标、打印机、网络适配器、硬盘、flash驱动器、便携式设备驱动器(如记忆棒)、CD驱动器、麦克风等都是在微型计算机中使用的I/O设备实例。

关于CPU和存储器架构的创新促成了指令级并行(ILP)、多线程和多核处理器等技术,在集成电路(IC)技术上的创新使得CPU的速度从1986年的16.7MHz(Sun-4 Sparc)提升到2010年的3.33GHz(Intel Nehalem Xeon),而在计算机互连架构中的创新提升了系统整体性能。今日,一个共享存储器系统可运行多个应用程序,并允许程序与多种I/O设备同步通信。此外,在互连结构中的创新生成了“即插即用”I/O设备接口,使得现代微型计算机用户可便捷地使用大量的I/O设备。

所有的I/O设备并不是以同样的方式进行运转的,各个设备的频率、速度、通信数据量的大小都是不同的。此外,在某些情况下,处理器必须直接参与某些存储器的通信过程。在另外某些情况下,当系统中的某些设备需要服务时,处理器需要轮询设备以提供服务(如发送和接收数据),或者设备通过中断的方式告知处理器其需要服务。从专用目的处理器到存储器单元、I/O设备等多种系统组件的运转方式是不同的,并且,与处理器相比其运行速度也多种多样,需要使用特殊的硬件模块,将这些组件与一个或多个处理器互连起来。

存储器控制器控制时序并响应存储器读写请求。设备控制器接口(DCI)已在第1章中介绍,其可以是一个简单的也可以是一个复杂的嵌入式系统,在处理器和I/O设备以及处理器和存储器之间扮演一个“中间人”的角色。通过一个被称为桥的部件转换一个组件(如处理器)使用的通信协议为其他设备(如GPU和磁盘DCI)使用的典型的标准协议。

作为一个简单的嵌入式系统,一个DCI是一个典型的**微控制器**,如后文的论述,这是一个包含CPU、RAM、ROM以及用于接口的其他模块的小系统。每个I/O设备额外需要一个**设备控制器(DC)**,DC是一个微控制器,来控制设备硬件的相关动作。例如,一个键盘设备控制器控制键盘的硬件,磁盘控制器控制磁盘驱动器的硬件等。一个通过有线或无线连接到DC的DCI执行下面两个任务:

- 它通过将控制数据发送到DC来控制设备的功能。
- DCI通过与DC通信来与设备交换数据。其可以从一个输入设备(如键盘)中接收到数据,也可以发送数据到一个输出设备(如打印机),或者既从一个磁盘驱动器或网络适配器中接收数据,也向其发送数据。

此外,为了支持即插即用式设备,现代微型计算机使用一个或多个通用目的DCI,如一个与**通用串行总线(USB)**设备连接的接口通常被称为**主控制器接口**。USB主控制器可同时与多种类型的USB设备进行连接和通信。

在这一章，我们将讨论 4 代互连架构，从单总线到多总线架构，从集成架构到基于连接的点到点架构。本章将展示一系列 I/O 设备，着重说明其通信需求，并说明用于设计 DCI 和 DC 的 I/O 端口（在第 1 章中开始介绍）。

本章将讨论中断、中断结构和与存储器直接进行设备通信的相关要求，也展示了相关的电路模块。提供一个中断处理电路的 CPU 数据通路示例供读者熟悉中断处理机制，并熟悉 CPU 给设备提供服务的必要步骤。并讨论说明提升性能的可选择的中断结构。最后，为了使读者更好地理解主控制器接口执行任务的过程，本章将展示一个 USB 主控制器接口的内部组织结构和通信协议。

互连架构

单一总线系统的体系结构比较简单，所有系统模块共享单一的总线进行通信。然而，随着微处理器和存储器的速度出现分歧，单一总线架构不再能有效地工作。处理器工作的时钟频率比系统中其他模块要高。此外，因为处理器使用专用的总线来与处理器和其他设备进行通信，设备制造商需要使用一系列的标准 I/O 总线来连接 DCI 与存储器或处理器。反过来，这造就了几代多总线系统架构。

为了简化设计并降低个人计算机的成本，包含存储器控制器、特定的桥和 DCI 在内的多个模块被集成到两个 IC 中：一个 IC 芯片是用于连接（如处理器或可选的 GPU）快速模块到存储器，另一个 IC 芯片是用于连接多种标准 I/O 总线接口的 DCI。然而，虽然集成互连架构在处理器核心有限的情况下可以工作得很好，但是随着处理器的数量和每个处理器中核心的数量的增加，这一架构会有一个存储器瓶颈。

这种集成的互连网络也不是可扩展的，不可以使用更多的存储器单元来增加存储器带宽。因此，需要一种可扩展的互连架构，允许在不显著增加存储器延迟的情况下增加存储器带宽。

1. 单总线

图 9-1 阐述了一个单总线架构，其包含一个 CPU、一个存储器单元和 3 个用于与 3 个外围设备连接的 DCI。总线包括一个地址总线（AB）、一个双向数据总线（DB）和一个控制总线（CB）。CPU 将使用总线来访问存储器并和其他 DCI 通信。例如，CPU 将把寄存器中的内容作为数据或命令发送给 DCI，相应的，CPU 也可通过 DCI 接收来自 DC 的数据和状态信息。

总线也用于在磁盘驱动器和存储器之间传输大量数据（如读一个文件）。然而，为了使计算机按预期运转、正常运行程序、无延迟地接受用户的输入数据等，即使总线上正在进行存储器读取磁盘中内容或往磁盘中写内容等工作，CPU 依旧会使用总线来访问存储器。因为这个原因，另一个模块称为直接存储器访问（DMA）控制器，将被使用。磁盘 DCI 和 DMA 控制器都将从 CPU 接收命令，在磁盘和存储器之间传输大数据块，在实际传输数据过程中，CPU 将不涉入传输过程。CPU 和 DMA 控制器将共享总线，轮流访问存储器。然而，当前只有微控制器使用单总线结构。

2. 多总线

图 9-2 阐述了一个使用 mezzanine 总线等级的多总线系统架构示例 [1]。它使用一个前端总线（FSB）和一系列标准 I/O 总线的结合，如表 9-1 所示。对表 9-1 中的外围设备的处理

方式也不一样；它们将被分为慢速设备、中速设备和高速设备。在图中，标准 I/O 总线的等级用于将高速设备和常用的通信设备与慢速和不常用的设备区分开。

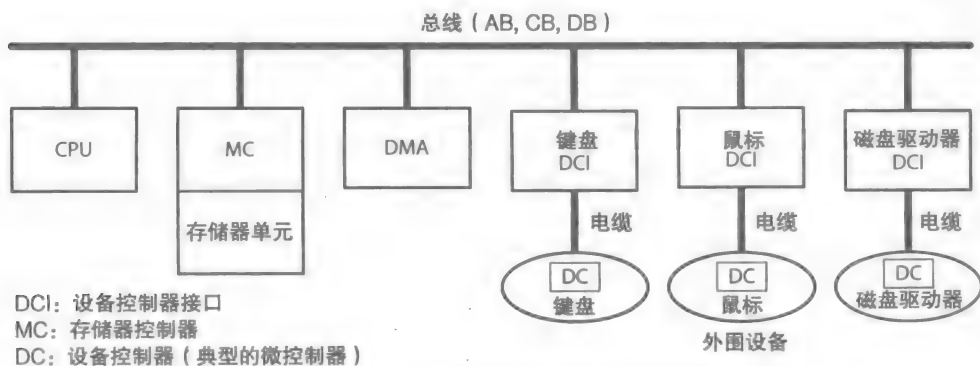


图 9-1 一个简单的微型计算机系统架构

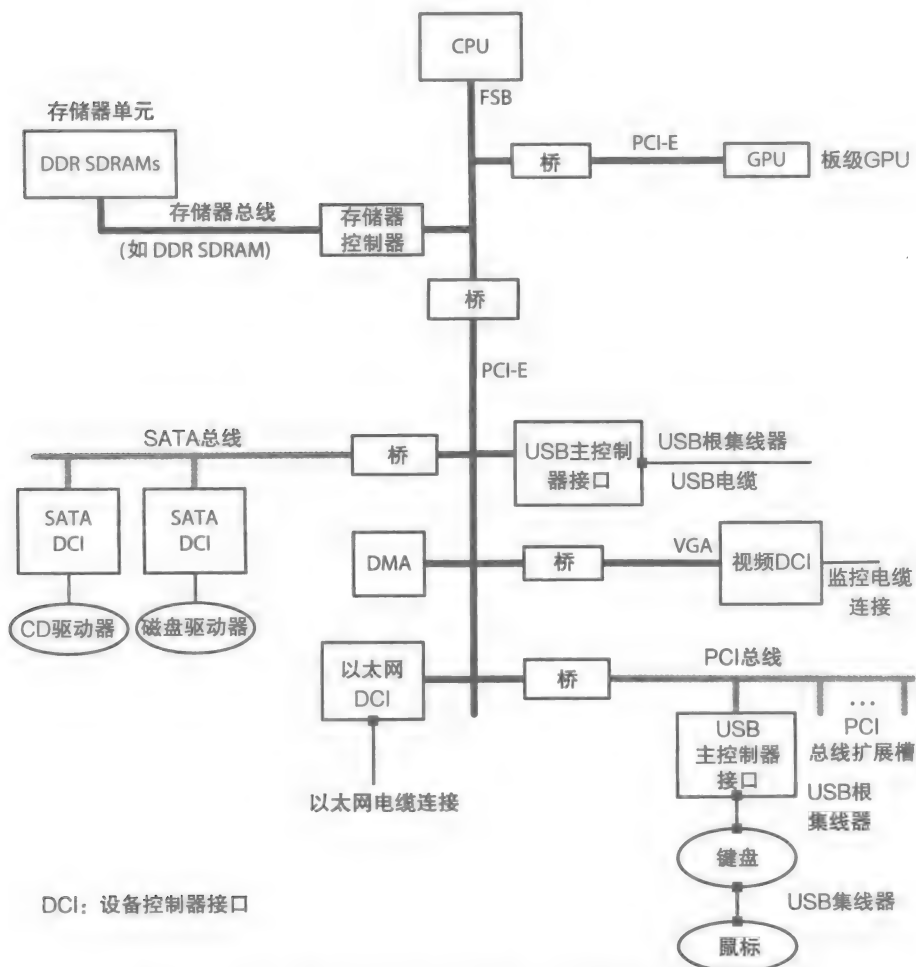


图 9-2 包含前端总线和 I/O 总线的多总线微型计算机架构

表 9-1 现代总线列表

| 名 称 | 描 述 |
|----------|---|
| AHB | 先进高性能总线 |
| APG | 图形加速端口, 包括 APG-2x, 4x 等 |
| ATA | 高级技术附件, 包括 ATA-1, 2 等, 也被称为并行 ATA (PATA) 或集成驱动电路 (IDE) 和串行 ATA (SATA) |
| DMI | 直接媒体接口总线, 包括 HDMI |
| DVI | 数字媒体接口 |
| FireWire | 火线 |
| PCI | 外围设备组件互连标准, 包括 PIC-bus、PCI-X、PCI 串行总线 (PCI-E 或 PCIe) 和紧凑型 PCI (CPCI) |
| SCSI | 小型计算机系统接口, 包括 SCSI-1, 2 等以及超级 SCSI |
| USB | 通用串行总线, 包括 USB1.x, 2.0 等 |
| VGA | 视频图形适配器 |

一个存储器控制器和两个 PCI-E 桥通过接口与 FSB 连接。桥是用于连接 GPU 并提升 FSB 与其他 I/O 设备通信的速度。在这种情况下, 与 I/O 设备连接的 PCI-E 总线通过一个以太网 DCI、一个 USB 主控制器接口、一个串行高级技术附件 (SATA) 桥、一个视频图形阵列 (VGA) 桥、一个 DMA 控制器、一个低速 PCI 总线扩展桥来连接系统中的其他部件, 并成为一个**母板**。

SATA 总线支持与 SATA 硬盘和光盘驱动器的链式连接。二级 USB 主控制器接口通过一个可用的 PIC 总线扩展槽显式安装, 与 USB 键盘和 USB 鼠标相连接。鼠标和键盘通过位于其中的 **USB 集线器** 连接系统。每个设备通过 **USB 端口** 与系统连接, 并通过一个或多个 USB 集线器和 **USB 根集线器** 与 USB 主控制器接口相通信。

3. 集成架构

通过使用互连芯片, 如被称为**北桥**的存储器控制中心 (MCH) 和被称为**南桥**的输入 / 输出控制中心 (ICH), 简化了 Intel 和 AMD 下一代计算机系统的板级设计, 如图 9-3 所示。原来分散在主板上的各模块被集成到 MCH 和 ICH 芯片中。在图中, MCH 包含一个存储器控制器、一个连接到可选的加速器 (如 GPU) 的桥和一个连接到 ICH 的桥。ICH 包含一个 DMA 控制器、一个 HDA 接口、一个 USB 主控制器接口、一个网络 DCI 和一系列与标准总线连接的桥。

377

存储器集线器后来扩展为可支持多个处理器运作, 生成了一种**统一存储器访问 (UMA)** 架构, 但这种架构只有在处理器核心 (CPU) 数量很少时 (如 4 个) 才有较高的效率。随着处理器核心的增加, 对存储器的访问增多, UMA 会导致存储器访问延迟变长。

4. 点对点架构

图 9-4 说明了一个基于 Intel 的**快速通道链接**或 AMD 的**超传输**或**隧道传输架构**的可扩展的**非同统一存储器访问架构**的示例。这个链接是用于在处理器之间生成一个**一对一**的互连网络, 以提供连续的点到点的通信。处理器不再通过公用的 FSB 来访问存储器, 每个处理器都是直接访问其自身的存储器, 生成一个 **NUMA 架构**。正如第 7 章中所讨论的, 在 NUMA 架构中平均存储器访问延迟比 UMA 架构少。而且, 当系统中互连的处理器增多时, 在 NUMA 架构中, 平均存储器访问延迟只会缓慢增加。

此外，由于对单芯片嵌入式系统（片上系统或 SoC）的需求的增加，更多模块，如存储器集线器，将集成到处理器芯片中。例如，在 Intel 的 Sandy Bridge 架构中，一个单独的处理器芯片包含了一个存储器控制器、PCI-E 和其他桥。图 9-5 说明了一个双节点的 NUMA 系统。

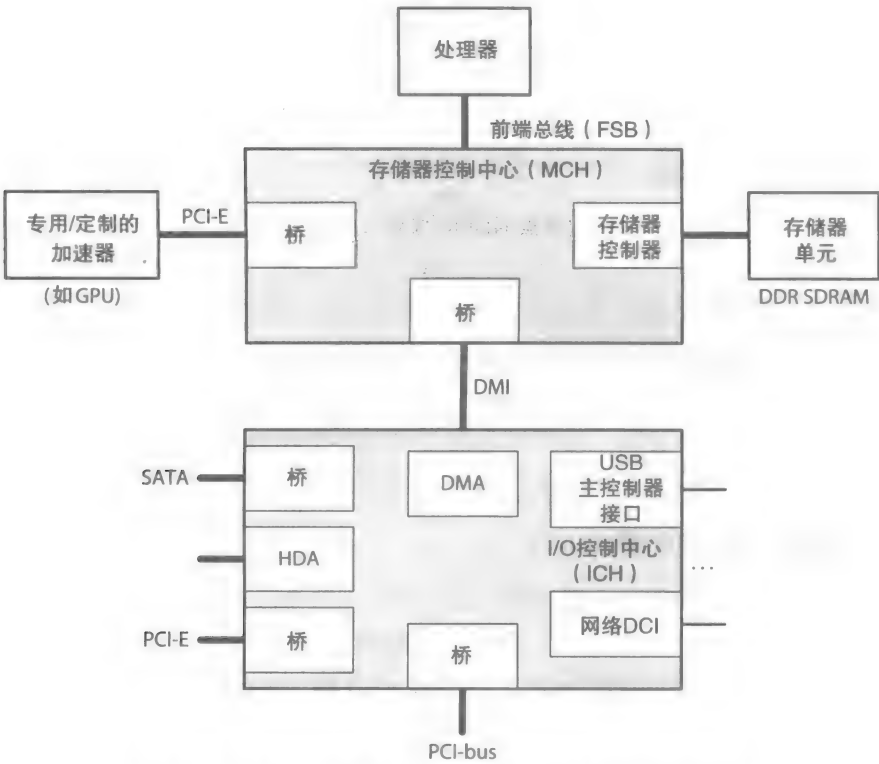


图 9-3 采用互连集成电路的集成微型计算机系统架构

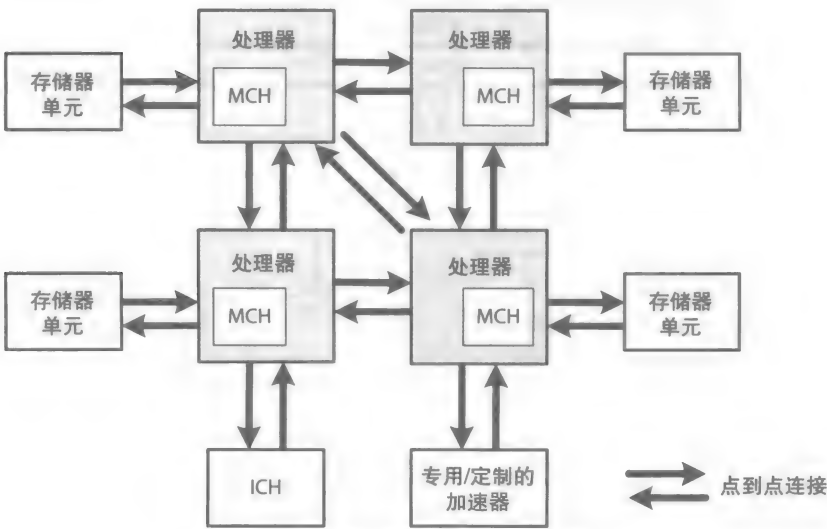


图 9-4 现代微型计算机 NUMA 架构

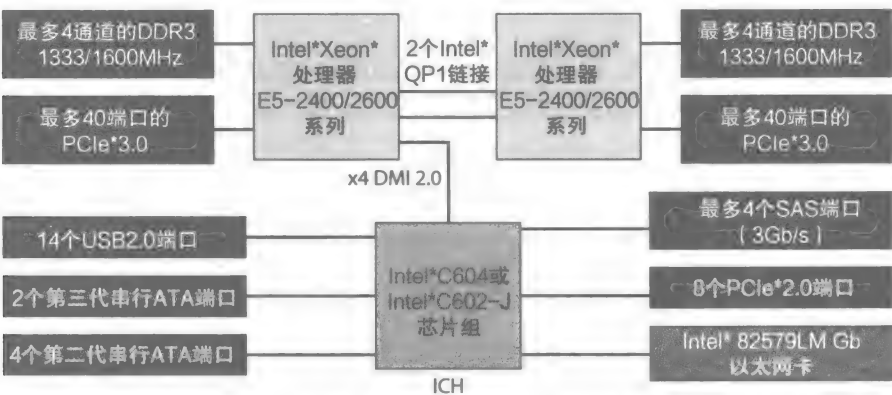


图 9-5 双节点 NUMA 系统 (由 Intel 提供)

9.2 存储器控制器

存储器控制器负责响应存储器的请求操作。它的复杂性取决于所使用的处理器和内存单元的通信协议。例如，一个简单的包含静态随机存取存储器 (SRAM) 的单总线架构要求一个简单的存储器控制器。另一方面，使用如 Intel 处理器的 FSB 的复杂总线的现代系统与设备，如同步动态随机存取存储器 (SDRAM)，通信时，将需要一个更加复杂的存储器控制器。

9.2.1 简单的存储器控制器

图 9-6 阐述了一个简单的存储器控制器的示例。控制器包含一个计数器和一个组合电路 (CC)。处理器控制总线包含 4 个信号，标记为地址选通信号 (_as)、写信号 (_wr)、读信号 (_rd) 和确认信号 (如 ack)。当信号 _as 选中时，将开始一个存储器读或写周期。_as、_wr 和 _rd 信号是从处理器视角的典型的存储器控制信号，而 _ce、_we 和 _oe 信号 (第 7 章中介绍) 是典型的从存储器视角的控制信号。

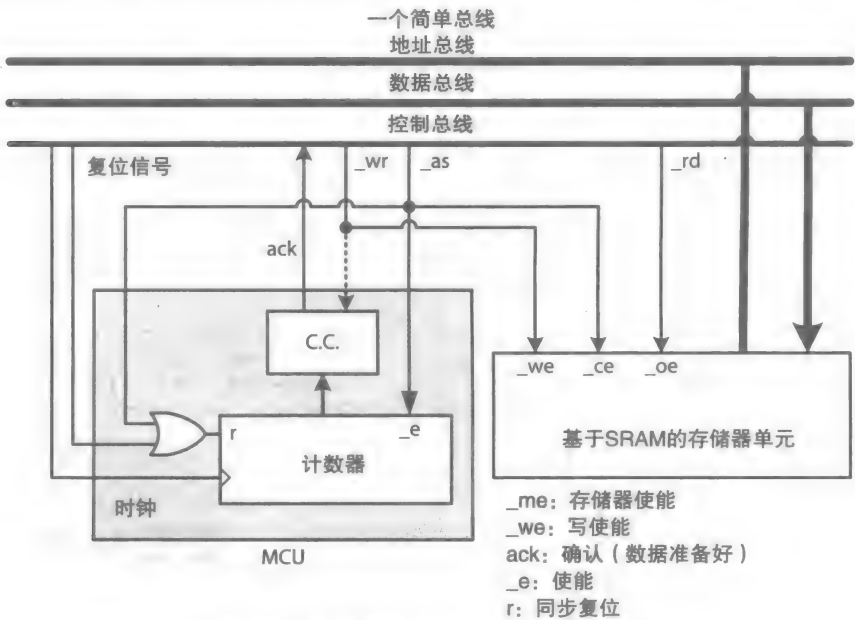


图 9-6 处理器视角的 SRAM 控制单元

如果存储器有多个存储器单元，_as 信号和目标存储器地址将用于生成 _ce 信号从而确定访问哪个存储器单元。在图中，假定了这个系统只有一个存储器单元，因此 _as 信号直接与 SRAM 存储器单元中的 _ce 信号相连。

正如图中所示，当 _as 选中时，存储器单元和计数器都将使能。计数器一旦使能，将在每个时钟周期递增，并当 _as 信号变为无效或者主机 reset 信号选中时，计数器将异步复位。计数器是用于统计访问存储器所需的等待周期数，等待周期数与存储器访问时间成正比。计数器模块维持一个 ack 信号，通过 ack 信号通知处理器完成了一个存储器读或写操作。当处理器等待 ack 信号被选中时，处理器被称为处于等待状态（也被称为空闲状态）。

380

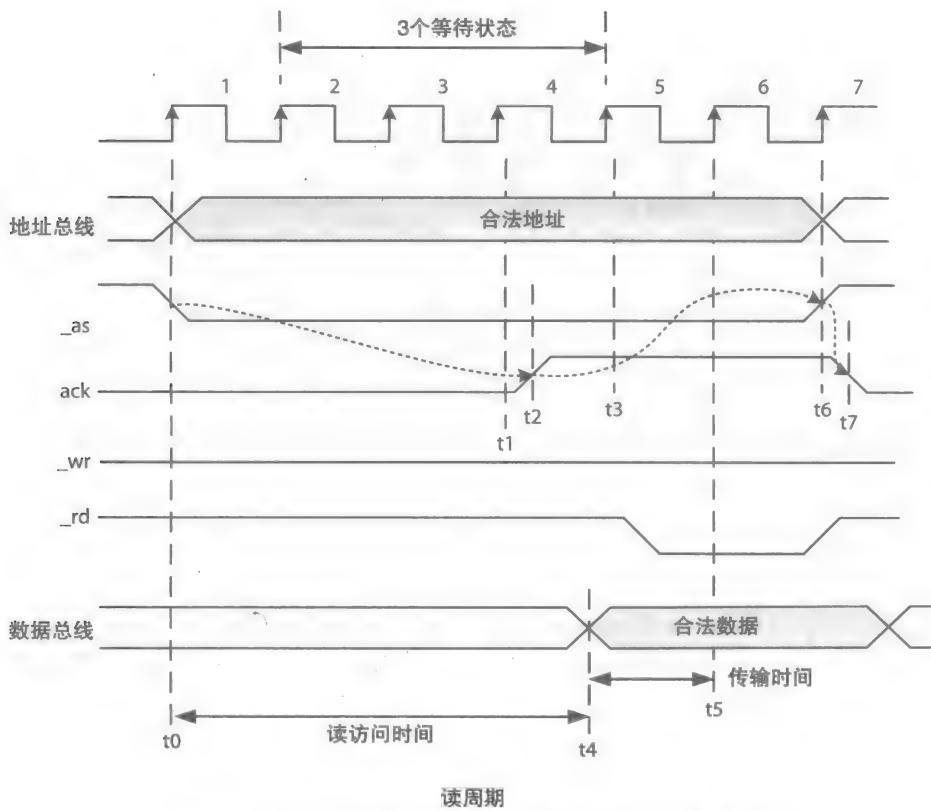


图 9-7 从处理器视角阐明了一个简单的 SRAM 读周期

图 9-7 从处理器视角阐明了一个存储器读周期。处理器等待周期数取决于公式 (9-1)。标记 $\lceil \cdot \rceil$ 表示向上取整函数， τ 为总线的时钟周期， m 为处理器检测到 $ack = 1$ 并结束该存储周期所需要的时钟周期数。例如，假设图中的时钟周期为 10ns，存储器读周期为 45ns，处理器需要一个时钟周期来检测到 $ack = 1$ ，需要另一个周期来结束读周期。根据公式 (9-1)。等待周期数为 3 ($\lceil 45ns/10ns \rceil - 2 = 3$)。

$$\text{等待周期数} = \left\lceil \frac{\text{Max (读访问时间, 写访问时间)}}{\tau} \right\rceil - m \tag{9-1}$$

在图中，三条带箭头的虚线表示信号依赖关系，通常被称为信号握手。处理器与存储器控制器通信起始于处理器将目标地址放置到地址总线上并选中 _as 信号，这表示存储周期开

381

始于 t_0 时刻。 $_as$ 信号变为有效电平将使能计数器和存储器单元。当存储周期开始后，第 4 个时钟周期 t_1 时计数器的值变为 3。并导致 ack 信号在 t_2 时（图 9-6 中， $\Delta cc = t_2 - t_1$ ）变为 1。这通过图中从 $_as = 0$ 到 $ack = 1$ 的箭头说明。

当处理器在 t_3 时检测到 $ack = 1$ （一个时钟周期之后），在下一个时钟周期，即 t_5 时，处理器从等待状态转为准备好状态，加载数据到其内部寄存器中。如图所示，存储器从 t_4 （ $45ns = t_4 - t_0$ ）时开始，将数据放置到数据总线上。处理器在 t_6 时，将 $_as$ 信号置为无效，使之变为 1。在 t_7 时， ack 信号电平变为 0，为无效电平， $_as$ 信号为 1，计数器复位，处理器和存储器之间的通信结束。在图中，这一过程是通过从 ack 变为 1 到 $_as$ 信号变为 1 和从 $_as$ 信号变为 1 到 ack 信号变为 0 的两个带箭头的虚线阐述说明的。下一个存储周期可从第 8 个时钟周期或之后开始。

存储器的写周期与读周期过程相似，不同之处在于当处理器检测到 $ack = 1$ 时，可将数据从数据总线上移除。再次，我们假定处理器需要一个时钟周期来检测 $ack = 1$ ，并需要另一个时钟周期来将数据从数据总线上移除，正如我们在读周期中做的那样， ack 信号可能更早被选中。此外，如果存储器读访问时间和写访问时间不同，存储器控制器可能实现两个不同的等待周期，一个基于存储器读访问时间，另一个基于存储器写访问时间。

对于一个动态随机存取存储器（DRAM），与 SRAM 中讨论的控制器相似，其存储器控制器也需要一个计数器。此外，DRAM 存储器控制器需要一个电路来实现刷新周期。然而，当一个刷新周期触发时，典型的，存储器控制器将允许当前的读或写周期（如果有）在刷新周期进行之前完成操作。一个新的读 / 写周期只能在正在进行的刷新周期完成后开始。

9.2.2 现代存储器控制器

图 9-8 说明了一个与复杂的 FSB 和当代的存储器单元相连接的存储器控制器。该控制器解释从复杂总线上接收到的存储器请求并通过标准 DDR SDRAM 总线与一个双速率（DDR）SDRAM 进行通信。表 9-2 说明了移动 Intel 奔腾 4 处理器的 533MHz FSB 的部分信号。一个 FSB 或者一个复杂的串行链接，例如 Intel 的快速通路，采用分离事务的方式与一个存储器控制器进行通信。一个分离事务包括一个事务请求和一个事务响应。

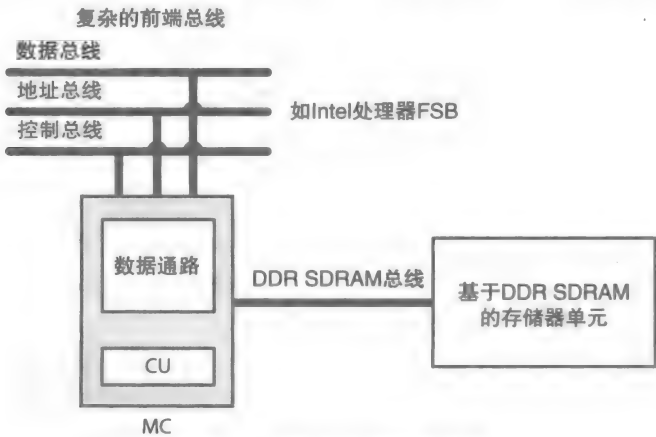


图 9-8 复杂的存储器控制器

表 9-2 移动 Intel 奔腾 4 处理器的 533MHz FSB 的部分信号

| 信号名称 | 总线 | 功能 |
|-------------|----|---|
| A[35:3]# | AB | 地址总线 (AB) |
| D[63:0]# | DB | 数据总线 (DB) |
| REQ[4:0]# | CB | 请求总线命令 (定义了总线事务类型) |
| ADS# | CB | 地址选通, 表明 AB 上地址是合法的 |
| ADSTB[1:0]# | CB | 通过目标模块, 地址选通 0 将锁存 A[16:3] 和 REQ#[4:0], 地址选通 1 将锁存 A[35:17] |
| AP[1:0]# | CB | 地址校验位 |
| DBI[3:0]# | CB | 数据总线反转 (16 位一组, 说明 DB 信号的极性) |
| DP[3:0]# | CB | 数据校验位 |
| DRDY# | CB | 数据准备好信号 |
| DSTBN[3:0]# | CB | 数据负边沿选通 (用于从 DB 中载入数据, 16 位一组) |
| DSTBP[3:0]# | CB | 数据正边沿选通 |
| RS[2:0]# | CB | 响应状态 (完成事务的状态) |
| RSP# | CB | 响应校验位 (通过校验保护 RS[2:0]#) |
| RESET# | CB | 处理器复位 |
| BCLK[1:0] | CB | FSB 时钟差分信号对 (2 个时钟信号彼此相反) |
| BINIT# | CB | 总线初始化信号 (如复位总线仲裁状态机) |
| BNR# | CB | 阻止下一请求 (挂起总线) |
| BPRI# | CB | 总线优先请求 (用于其他设备请求访问 FSB) |
| BR0# | CB | 总线请求信号, 用于处理器请求总线访问 |
| DBR# | CB | 数据总线复位 (用于调试) |
| DBSY# | CB | 数据总线忙 (表明 DB 正在被使用) |
| DEFER# | CB | 推迟事务 (当 active 表明总线事务无法完成) |
| HIT# | CB | Snoop 命中 (用于 cache 访问一致性) |
| HITM# | CB | Snoop 命中修正 (如果与 HIT# 同时使用, 表明需要一个 snoop 停顿) |
| LINT[1:0] | CB | 本地 APIC (高级可编程中断控制器) 中断信号 |
| LOCK# | CB | 表明原子的总线事务, 确保没有新的请求事务被允许到相同的存储单元, 直到这一存储单元中的未完成的请求事务的响应事务完成 |

382
?
383

一个读请求事务包含一个地址和一个命令 (如读)。一个写请求事务还包含了数据。一个响应事务包含一个或多个确认信号, 当响应一个读请求时, 响应事务还包含数据。一个典型的现代存储器控制器可处理多个未完成的分离事务。

在表中, 标记 # 是用于表示低电平使能信号。使用 533MHz FSB 的移动 Intel 奔腾 4 处理器 [2] 有 478 个管脚, 其中部分信号因为将用于与电源或地相连接、电源管理、板级设计问题、性能测试、总线错误信号、事务流控制和测试等工作而保留。CPU 中有 36 个管脚用于地址总线 (A[35:3]#), 其可访问 2^{36} 字节的物理空间, 有 5 个管脚用于解决存储器命令 (REQ[4:0]#), 64 个管脚用于数据总线 (D[63:0]#)。一个请求事务 (如地址和命令) 通过地址选通信号来解决, 其中地址选通信号 (ADS#) 是由存储器控制器在 2 个部分中使用 2 个额外的选通信号 (ASTB[1:0]#) 来锁存。ADS# 用于启动或结束一个存储器请求事务, ASTB[1]# 和 ASTV[0]# 用于控制时钟信号, 来加载目标地址的高位和低位数据到位于存储器控制器中的两个寄存器缓冲区中。

响应事务包含一个数据准备好信号 (DRDY#), 一个 3 位响应码 (RS[2:0]#), 以及响应读命令的数据。如果响应事务包含数据, 它将被处理器在数据选通信号的 4 个负边沿 (DSTBN[3:0]#) 或正边沿 (SDTBP[3:0]#) 锁存。36 位的地址、64 位的数据和 3 位的响应码也都包含奇偶错误校验位, 其中 AP[1:0] 用于校验地址, DP[3:0] 用于校验数据, RSP# 用于校验响应码。表中所列的剩余控制总线信号是用于处理 cache 存储器 (第 10 章中讨论)、外围设备的服务请求以及当有两个或多个 CPU 试图修改共享存储器中某地址内容时的原子总线访问。同一时间只允许一个 CPU 修改共享存储器中的内容。

现代存储器控制器在处理器和存储器之间担任中间人的角色, 其通过 FSB 连接 CPU, 另一端与使用 SDRAM 模块的存储器相连。处理器通过使用, 如表 9-2 中的 A、D、REQ、ADS 和 ADSTB 总线信号, 来发出一个向存储器控制器请求访问存储器的事务。控制器将记录该事务, 并转换 FSB 的通信协议为 SDRAM 总线上使用的协议, 反之亦然。可参考第 7 章中 SDRAM 总线事务的示例。

9.3 I/O 外围设备

每一个外围设备都要求一个专用的通信协议来控制速率和数据大小。例如, 键盘在某个按键按下时将生成一个小的数据 (如一个按键值)。而磁盘驱动器是用于快速传输大量数据的机电装置, 磁盘有多个用于记录的柱面, 每个柱面有多个磁道 (和 CD 上的磁道相似)。每个磁道被分为多个相等大小的区域, 这个区域叫作扇区。每个柱面有其自己的读/写头, 某些磁盘可同步从多个扇区中传输数据。

384

例如, 考虑 Samsung 260GB 的硬盘 (如 HD642JJ)。这一磁盘转速为 7200 转/分 (RPM), 在与 16MB 的内置存储器 (DRAM) 之间, 峰值传输速率为 175Mbps (兆位每秒)。内置存储器, 也被称为缓冲区, 是用于临时存储从主存储器中读出或向主存储器中写入的扇区数据。在缓冲区和主存储器之间的峰值传输速率为 300Mbps。其扇区大小为 512B。现代个人计算机使用扇区大小为 4KB 的磁盘, 某些现代磁盘驱动器可以运转在 10 000RPM 或更高的速率之上。

Samsung 硬盘驱动器平均查找时间为 8.9ms (毫秒), 其为移动磁盘读/写头到一个特定磁道上所需要的时间。磁盘驱动器的平均延时为 4.17ms, 其为旋转磁盘 50% (磁道周长的一半) 所需要的时间。这就是说, 当磁盘转速为 7200RPM, 旋转一半 (50%) 所需要的时间 (0.5/7200RPM)。两者时间之和为磁盘进行读或写操作时确定并移动到目标扇区所需要的近似时间。

例 9-1 使用 Samsung 磁盘驱动器的平均传输时间和峰值传输速率等性能参数, 确定传输 512B 数据 (数据在一个扇区内) 到存储器所需要的近似平均时间。

解: 将扇区数据复制到存储器中需要完成 4 个连续的任务:

任务 1: 搜索目标磁道, 其平均查找时间为 8.9ms。

任务 2: 确定目标扇区, 其平均延迟为 4.17ms。

任务 3: 从扇区中复制数据传输到内置存储器中, 其传输的峰值速率为 175Mbps。

任务 4: 从内置存储器中复制数据到主存储器中, 其传输的峰值速率为 300Mbps。

从磁盘扇区传输 512B 数据到主存储器中的平均近似时间为 13.075ms, 计算公式如下:

$$\begin{aligned}
 T_{512B} &= 8.9\text{ms} + 4.17\text{ms} + \frac{512\text{B}}{175\text{Mbps}} + \frac{512\text{B}}{300\text{Mbps}} \\
 &= 13.07\text{ms} + 0.0029\text{ms} + 0.0017\text{ms} \\
 &= 13.075\text{ms}
 \end{aligned}$$

然而，从磁盘往存储器或从存储器往磁盘中传输多个物理上连续扇区的数据将更快。传输过程与从存储器往磁盘中传输数据的方式相似，唯一不同的是在数据写到目标扇区之前，其需从存储器中复制数据到内部缓冲区。

RAID（独立磁盘冗余阵列）的设计是用于提升磁盘的带宽和可靠性。例如，RAID 中数据采用交叉存取的方式，与存储器中交叉存取（第 7 章）相似，通过将一个单条记录的数据存储到多个扇区上，每个独立的磁盘上存储一片数据，从而提升 RAID 的带宽。数据交叉存储可以在位级（RAID-2 和 RAID-3）或在块级（RAID-0、RAID-4 和 RAID-6）使用。可重复使用文件的副本从而实现文件存储冗余（RAID-1）。错误校验码可以用于避免存储冗余所需的额外消耗，其在一个或两个磁盘失败，如在 RAID-2 到 RAID-4 中一个磁盘失败，或在 RAID-6 和 RAID DP（DP：双重奇偶校验）中两次磁盘失败的情况下，实现了恢复机制。

[385]

其他没有机械部件的设备可以以更高的速率进行通信。例如，以太网适配器可在两个通信的计算机之间以 10Mbps、100Mbps 或更高的速率传输数据。在这种情况下，数据来自于存储器，并通过 DMA 控制器和该适配器的 DCI 来与接收数据的计算机进行通信。接收数据的计算机通过它自己的 DCI 和 DMA 控制器来接收数据，并传输数据到主存储器中。

9.4 控制和连接 I/O 设备

外围设备有慢速、中速和高速 3 种，每种设备都需要一个不同频率的时钟，并以不同的传输速率传输数据。一些设备可能包含机械设备，其运转方式与数字系统不同，可能要求将数据转换为数字信号。在一个设备中，数据格式和信号的电平与其在处理器或存储器中使用的并不相同。此外，处理器可在不影响其他设备功能的前提下与各个设备进行通信。

一个 DCI 通过处理器总线（如图 9-1 所示）、I/O 总线（如图 9-2 所示）或 ICH（如图 9-3 或图 9-4 所示）与系统中的其他设备进行通信。在过去，个人计算机采用每个设备均有其专用的 DCI 的设计。即使是最基本的设备，如键盘、鼠标和打印机，都有其专用的 DCI，如图 9-1 所示，其中包含键盘和鼠标的 DCI。

例如，旧式标准中，如并行端口（IEEE 1284）和 RS-232（232 推荐标准），包含其小版本 DE-9，其仅支持单独的点对点连接来连接一个外围设备，并需要一个专用的 DCI。这表明其不支持“即插即用”设备接口，这增加了个人计算机的成本，此外，其每次安装一个新的设备时都需要重新启动计算机。一个典型的系统只能支持有限数量的插槽以供安装新设备，这也给定了个人计算机用户某些限制。今天，大多数外围设备（但并不是所有的）是即插即用的。

正如早先讨论的那样，DCI 和 DC 都使用 I/O 端口。端口采用三态缓冲门和寄存器设计，每个寄存器通过特定的地址来确定，其访问方式和存储器相似。独立寻址的 I/O 或端口映射 I/O 和存储器映射 I/O 是两种常用的 I/O 端口地址映射模式，如图 9-9 所示。

在图 9-9a 中，端口映射的 I/O 要求两个独立的地址空间，一个用于存储器（如 512B），另一个用于相同大小的 I/O 端口。并添加一条额外的控制总线信号，如 `_m`，来说明给定的地址是存储器地址或 I/O 端口地址，在 `_m = 0` 时是存储器地址，在 `_m = 1` 时是 I/O 端口地址。

另一方面，存储器映射 I/O 要求将一个地址空间分割为存储器和 I/O 端口两部分，如图 9-9b 所示。在图中，一个 512B 的地址空间被划分为存储器地址和 I/O 端口地址。控制信号（如 `_as`、`_rd`、`_wr` 和 `_ack`）既可以用于访问存储器，也可以用于访问存储器映射的 I/O 端口。

[386]

在这种情况下，由存储器控制器和 DCI 来确定一个地址是在存储器单元的地址范围内还是在 I/O 端口地址范围内。

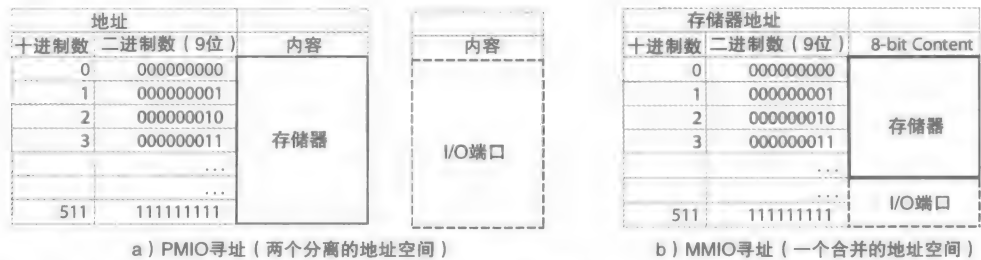


图 9-9 I/O 端口寻址方式：a) 端口映射 I/O，一个 512B 地址空间用于存储器，另 512B 地址空间用于 I/O 端口；b) 存储器映射 I/O，一个单独的 512B 地址空间划分为存储器和 I/O 端口两部分

任何计算机系统均可实行存储器映射 I/O 寻址模式。Intel 处理器也可支持端口映射 I/O 寻址，其包含两条特殊的 I/O 指令：“IN”和“OUT”。一般情况下，所有的精简指令集计算机（RISC）处理器支持存储器映射 I/O 寻址模式，这使其拥有直接访问存储器指令来访问 I/O 端口的优点。另一方面，端口映射 I/O 寻址方式拥有可分配任意存储器地址空间给 I/O 端口的优点；然而，这些年来，随着存储器地址空间的不断增长，这种优势已经消失殆尽。例如，现代计算机系统主存储器空间在 GB 级，存储器 I/O 端口映射方式有绝对足够的空间来访问任意多的 I/O 端口。

I/O 端口

图 9-10 说明了一个包含一个输入端口和一个输出端口的存储器映射 I/O 端口的简单示例。端口指的是用于在数据总线上使输入数据隔离的一组三态缓冲器和一个用于保存输出数据的并行输入输出的寄存器。在图中，并行输入输出的寄存器也被称为缓冲器，其由正极门栓构成。

[387]

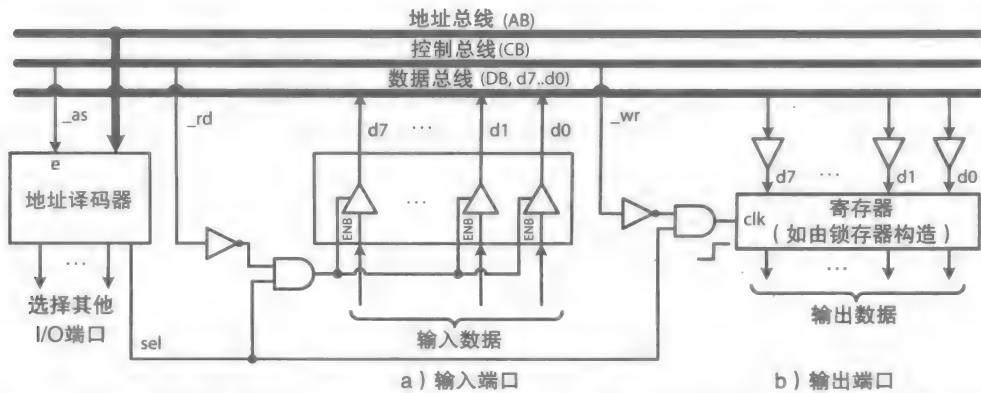


图 9-10 包含一个输入端口和一个输出端口的 I/O 端口示意图，假定采用存储器映射 I/O 端口寻址方式

对于输入端口，一个读周期以处理器将端口地址放置到地址总线上并选中 _as（地址选通）信号为开始，如图 9-11 中 t0 所示。当 _as 信号被处理器选中时，这表明现在位于地址总线（AB）上的地址是合法的。_as = 0 将使能图 9-10 中的地址译码器，此地址译码器之

后将选中地址总线上目标端口地址对应的 sel 信号。_rd 信号在 t2 时的 1-0 转换将使能三态缓冲器，导致输入数据在 t3 时放置到数据总线上。处理器将接收数据并在 t4 时解使能 _rd。在 t5 时，_as 变为 1 并且 sel = 0，读周期结束。

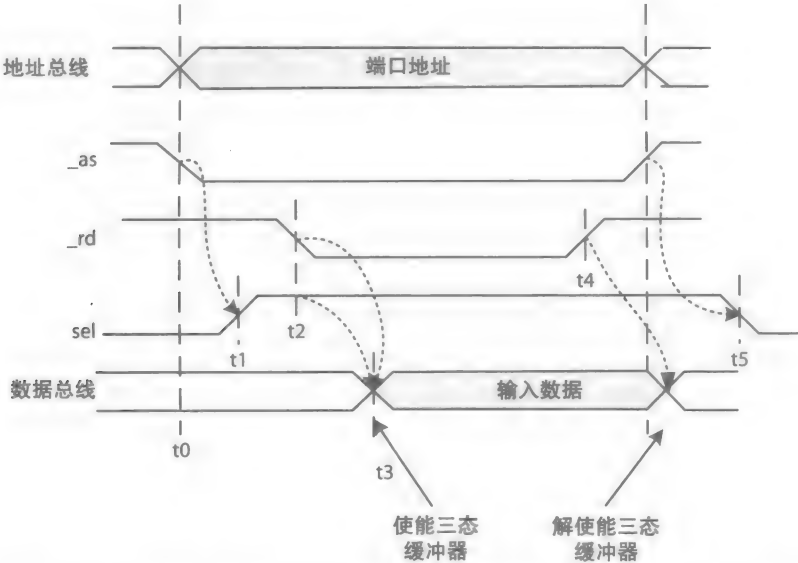


图 9-11 处理器视角的一个存储器映射 I/O 输入端口读周期示意图，也要求一个 ack 握手信号（未显示）

如图 9-12 所示，一个输出端口的写周期开始部分与输入端口的读周期相似。处理器将端口地址放置到地址总线上，并选中 _as 信号。当 sel = 1 时，_wr 信号的 1-0 转换将导致与寄存器相连接的与门输出端的 0-1 转换，如图 9-10 所示。这将导致正极并行加载寄存器（由正极锁存器构成）在 t1 时加载数据总线上的数据。_wr 信号的 0-1 转换将完成写操作。当 _as 信号在 t2 时变为 1，并且在 t3 时 sel = 0，写周期结束。

采用端口映射的 I/O 端口其读周期和写周期是相似的，不同之处在于在图 9-10 中地址译码器还需要输入控制总线信号 _m（之前已经说明）。假设一个累加器 ISA 处理器，其“LD”和“ST”指令（之前在第 8 章中讨论）将生成 _m = 0 来访问存储器。两条新的指令，例如，“IN”和“OUT”，将生成 _m = 1 来访问端口映射的 I/O 端口。而在采用存储器映射的 I/O 端口并不需要 _m 信号，“LD”指令和“ST”指令既可以访问存储器，也可以访问 I/O 端口。

1. 可配置端口

如图 9-13 所示，一个可配置端口也包含一个数据方向寄存器（DDR），其用于配置 I/O 端口中的每一位为 1 位的输入端口或 1 位的输出端口。图 9-14 说明了 1 位可配置 I/O 端口的设计细节。当 DDR.q₀ = 0 时，其通过不使能三态缓冲器 1 配置 d₀ 为 1 位输入端口。并在一个读周期中，三态缓冲器 2 将被使能。当 DDR.q₀ = 1 时，其通过使能三态缓冲器 1 配置 d₀ 为 1 位输出端口。三态缓冲器 2 一直不使能。

例如，当 DDR 中的内容为 0x0F 时，其配置图 9-13 中的 I/O 端口的高 4 位为 4 位输入端口，低 4 位为 4 位输出端口。DDR 中的内容可被读取并可被动态修改，从而在 I/O 端口设置时重新配置 I/O 端口。当 DDR 中的内容被读取时，三态缓冲器 3 将被使用。

然而，它也可能包含，例如，一个位于微控制器中的“命令”端口，这个端口可用来通过有限的配置选项来配置一系列 I/O 端口，足以支持多个 DCI 和 DC。

图 9-15 显示了一个包含 3 个可配置 I/O 端口，即端口 0、端口 3 和端口 4，以及两个多种目的 I/O 端口（端口 1 和端口 2）的示例。端口 1 和端口 2 不仅仅是可配置的，其还有多种用途。在一个应用程序中，与端口 1 和端口 2 相关的管脚可能用于配置 I/O 端口，在另一个程序中，同样的管脚也可用于数据信号或控制信号。

作为一个嵌入式系统，一个微控制器包括 CPU、RAM、ROM、一系列 I/O 端口、一个或多个定时器模块、一个中断控制器以及一个或多个数据通信模块等。当微控制器周期性地执行特定的任务时，将需要定时器模块。中断控制器将在后文讨论，当有一个外部事件时，中断控制器用于中断微控制器。

电擦除可编程只读存储器（EEPROM）和 flash 存储器（组织成存储器）是供嵌入式设备中的固件和软件使用。EEPROM 是用于加载引导装载程序，而 flash 存储器是用于存储 DC 或 DCI 中的固件，其可在系统建立时被系统更新。RAM 是用于在系统执行过程中存储程序数据并 / 或存储配置数据。一个定时器模块包含一个计数器，定时器用于调度事件。例如，一个使用 12K 模式的计数器并使用 12MHz 时钟的定时器模块可用于监测键盘等硬件，每 1ms 监测一次键盘是否有键按下（ $12\text{ K cycles} / 12\text{ M cycles/s} * 1000\text{ms/s} = 1\text{ms}$ ）。

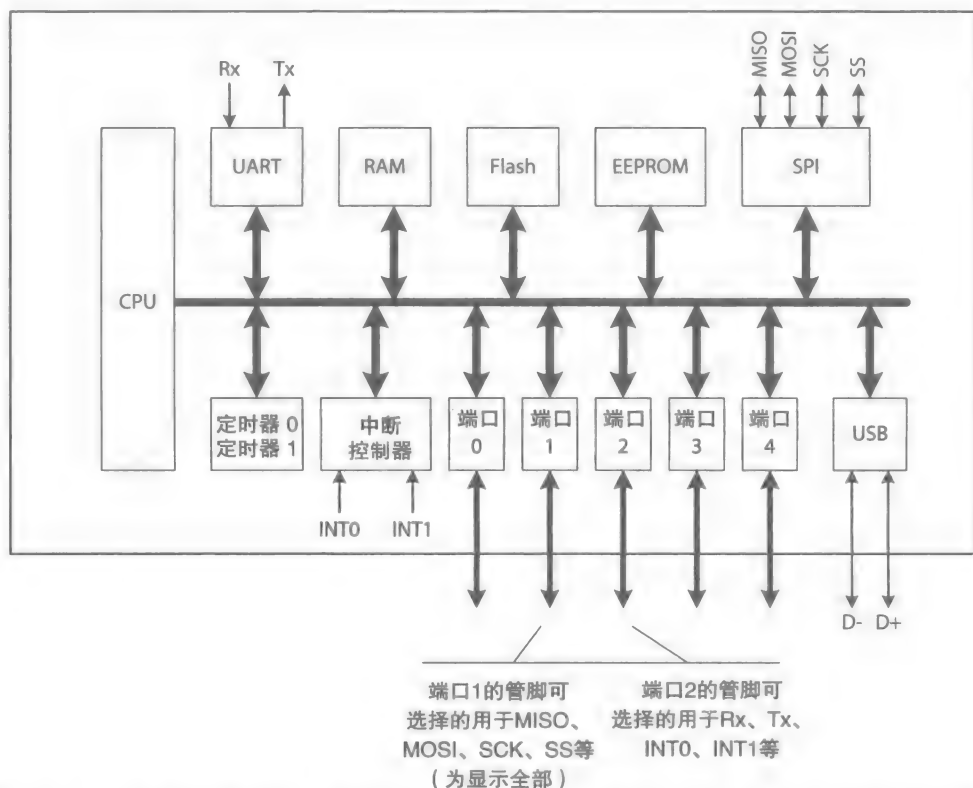


图 9-15 一个微控制器结构：连接到端口 1 和端口 2 的管脚有两种用途 [3]，未显示所有模块

390

在图中，微控制器也包含 3 了个不同的数据通信模块来支持多种设计程序。其包含一个通用异步收发器（UART）、一个串行外围接口（SPI）和一个 USB 端口，所有的模块都在一

个独立的芯片中。UART、SPI、USB 是三种不同的用于 I/O 设备的通信协议。

总体而言，一个微控制器中并不是所有的资源都将用于一个独立的应用程序，因此某些模块可能共享同一个 I/O 管脚。例如，在图中，端口 1 和端口 2 的管脚也可用作 UART 模块中接收 (Rx) 和发送 (Tx) 信号；也可用作 SPI 模块中的主输入从输出 (MISO)、主输出从输入 (MOSI)、同步时钟 (SCK) 和从设备选择 (SS) 信号或者中断控制器中的中断 0 (INT0) 和中断 1 (INT1) 信号。

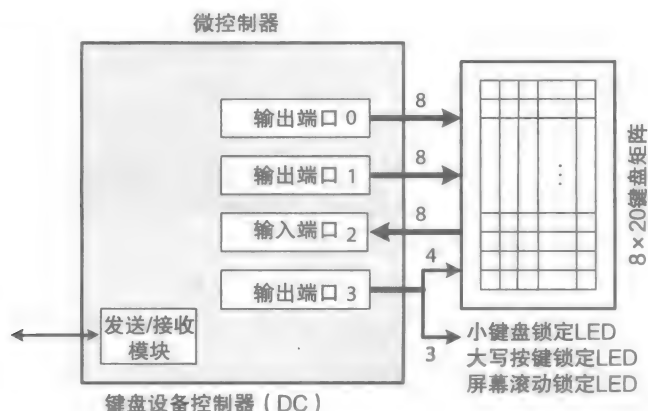


图 9-16 一个包含按键矩阵的键盘设备控制器，未显示所有的微控制器模块

图 9-16 说明了一个包含按键矩阵的键盘设备控制器。矩阵中每一行和列的交叉点可确定一个按键。设备控制器通过激活 20 个列信号来扫描矩阵，例如，通过使用端口 0、1、3 每次设置一条信号线为 0，如图所示，通过端口 2 读取 8 位行信号。当一个按键按下时，将连接一条列信号线和行信号线，只使得其对应连接的行信号和列信号为 0，其他行信号依旧为 1。当一个行信号返回为 1 时，表示该按键被释放。当某个按键被按下并建立联通时，DC 将生成一个**通码** (make code)，当一个按键被释放，联通关系解除时，DC 将生成一个**断码** (break code)。

键盘扫描码有 3 种标准，为**扫描码集合 1**、**集合 2** 和**集合 3**。例如，当使用扫描码集合 2 时，数据顺序为 0xE012、0xE01C、0xE0F01C 和 0xE0F012 表明大写字母 A。代码 0xE012 是 shift 键的通码，0xE01C 是 a 键的通码，0xE0F01C 是 a 键的断码，最后 0xE0F012 是 shift 键的断码。一个按键也可以被保留从而使得其通码可频繁使用。键盘 DC 通过**按键重复率**来初始化，按键重复率是通过计算一个键被按下时生成的通码产生的频率来决定的。

所有的按键，包括 CTRL、ALT 等，其通码和断码是通过 DCI 来传输并存储到存储器中。这个代码之后将通过基本输入 / 输出系统 (BIOS) 来转换为一个 ASCII 码，以供一个应用程序来使用。

键盘 DC 也通过键盘 DCI 来接收指令，例如，DC 在初始化过程中接收一个按键的重复率，或者当 CAPS-LOCK 键被按下时，如果有，接收一个点亮 LED 灯的指令。

2. 包含状态位的端口

图 9-10 中的 I/O 端口是一个不包含状态位的 I/O 端口示例。我们没有方式来确定什么时候输入数据是可使用的，什么时候输出数据可以被读取。无状态位的 I/O 端口用于直接控

制硬件设备，如图 9-16 所示，用于控制键盘矩阵的设备控制器采用以上设计。

另一方面，当需要表明输入数据何时可被 CPU 读取或输出数据何时可被其他设备读取时，要求 I/O 端口采用含有状态位的 DCI。例如，当与一个打印机 DCI 通信时，下一个需要发送给打印机打印的数据何时发送给打印机打印由已经接收到的数据的时间来决定。当 DCI 已经发送前一个数据给打印机 DC，并且 DCI 表明其已经准备好接收下一个打印数据时，下一个打印数据可以被发送给打印机 DCI。

类似的例子还有，比如，在键盘 DCI 中说明从键盘 DC 中接收到的新的扫描码是可使用的，或在 DMA 控制器中说明在主存储器和磁盘驱动器 DCI 或网络适配器之间的数据传输已经完成。

一个包含状态位的 I/O 端口样示程序将在下一节讨论，但是，首先，这里有三种不同的从 I/O 到 DCI 中传输数据的机制，其中一个 DMA 传输方式，这个在上文简略讨论过。

9.5 数据传输机制

DCI 之间的数据通信可直接由处理器或 DMA 控制器来完成。此外，当处理器直接与 DCI 通信从而传输数据时，数据事实上是存储在主存储器中的。处理器先从 DCI 接收数据，之后将数据写入存储器中；或者处理器先从存储器中读取数据，之后再从存储器输出到 DCI 中。中断驱动和程序控制 I/O 是两种处理器直接参与的数据传输机制。

9.5.1 中断驱动传输

中断驱动传输也可称为中断驱动 I/O，当设备产生一个中断时传输数据。当前正在运行的程序将停止，处理器转而处理一个中断处理程序，中断处理程序也被称为中断服务程序。当中断处理程序执行时，将在设备和主存储器之间传输数据。因为传输数据过程中，处理器正在执行中断处理程序，它直接参与了设备和存储器之间的数据传输过程。

中断结构（将在 9.6 节介绍）通过分配给每一个设备一个中断优先级以防止出现多个设备同时请求中断处理器的情况。某些设备（例如键盘）相对于磁盘驱动器而言具有较低的中断优先级。而当系统中有太多的设备时，对某些设备的服务可能会延迟。

为了说明中断驱动传输机制和包含状态位的 I/O 端口的应用程序，思考图 9-17 中一个传统键盘的 DCI。在图中，传统键盘的 DCI 拥有两个 I/O 端口，标记为端口 0 和端口 1。然而，因为处理器和 DCI 都将访问这些端口，一个供 CPU 读入数据的输入端口也可用作 DCI 的输出端口。相似的，一个供处理器写数据的输出端口也可用作 DCI 的输入端口。两个寄存器中的内容和端口地址相关，一个作为输入端口，另一个作为输出端口。

在图中，寄存器被标记为“扫描码”、“数据”、“状态”、“命令”。表 9-3 分别从处理器视角和 DCI 视角说明各端口为输入端口或输出端口。图中的端口是以键盘 DCI 视角来标记的。

处理器可以在任意时间读取“状态”端口，其包含了剩余端口的状态位。输入缓冲区满（IBF）状态位（或高电平使能信号 ibf）和输出缓冲区满（OBF）状态位（或低电平使能信号 _obf）表明了其他三个端口的状态。当 _obf = 0（满）时，这表明 DCI 已经加载一个扫描码到“扫描码”寄存器中，供处理器访问。当 ibf = 1（满）时，这表明从处理器发送来的数据已存入“数据”寄存器或“命令”寄存器中，供键盘 DCI 访问。

[392]

[393]

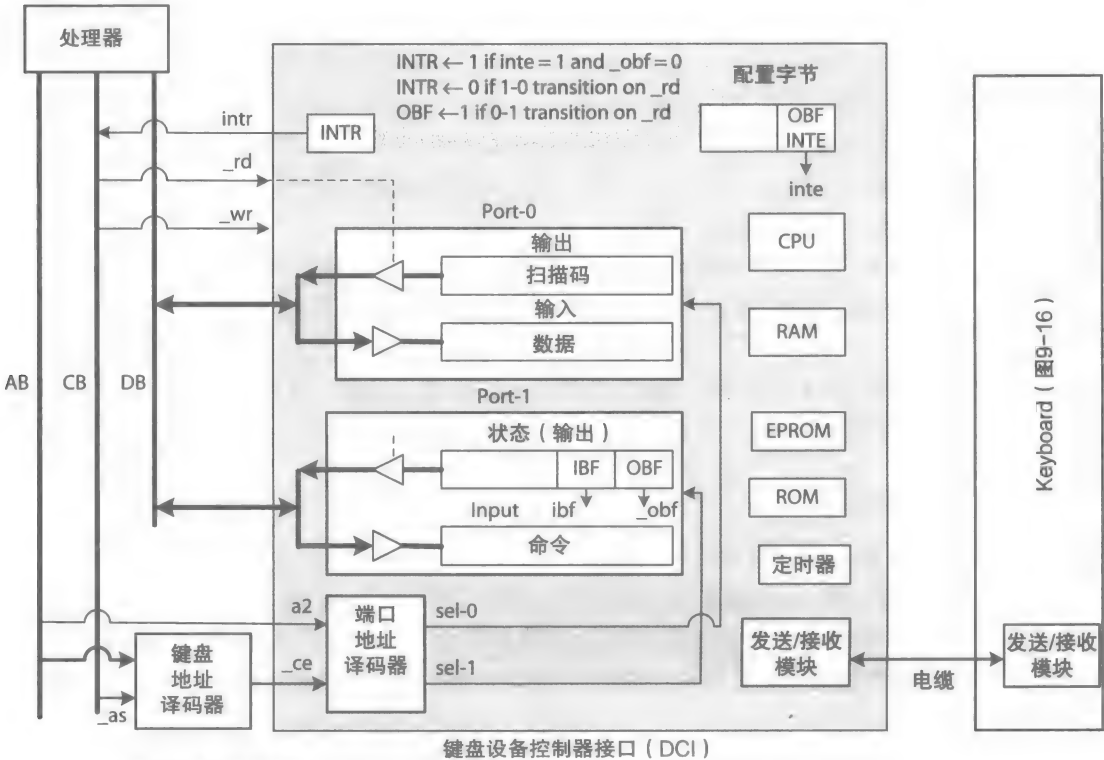


图 9-17 通过键盘 DCI (设备控制器接口) 来阐述 I/O 端口访问和中断驱动传输, 未显示所有的信号和数据通路模块, 端口是以 DCI 视角进行标记

表 9-3 从处理器视角和设备控制器接口视角看图 9-17 中的端口

| 端口名 | 从处理器视角 | 从键盘 DCI 视角 |
|-----|--------|------------|
| 扫描码 | 输入端口 | 输出端口 |
| 数据 | 输出端口 | 输入端口 |
| 状态 | 输入端口 | 输出端口 |
| 命令 | 输出端口 | 输入端口 |

一个**键盘驱动服务程序**必须先配置 DCI 为中断驱动传输或程序化数据传输。由处理器执行的驱动服务程序将写一个配置命令到“命令”寄存器中来初始化 DCI。DCI 接收该命令后, 将开始执行配置程序。驱动服务程序之后将写配置数据到 DCI 的“数据”寄存器中。DCI 的配置程序将读取该数据并存储到 DCI 中的“配置缓冲区”中。

例如, 为了说明 CPU 如何访问 I/O 端口, 我们通过下面用于检测 IBF 位的程序代码来说明, 当 IBF 位为 0 (如 ibf = 0), 表示配置命令已存储到“命令”寄存器中。该程序使用累加器 ISA 进行编写, 累加器 ISA 已在第 8 章中说明。

键盘驱动程序先检测“状态”端口, 再输出命令到“命令”端口:

```
...
...
LOOP: LD (Port_1)    //input keyboard status
      AND 2          //select status bit 1, ibf; bit 0 is _obf.
```

```

CMP 2          //is ibf = 1? 1 indicates full.
JEQ LOOP       //wait for an empty input buffer
LD ...         //get, configuration command
OUT (Port_1)    //output to "Command" port, assuming
                //port-mapped I/O port
...

```

对于中断驱动传输程序，配置数据必须配置“配置缓冲区”中的 OBF 中断使能位 (INTE)。当 INTE 使能时，中断请求位 (INTR) 将用于触发中断。这就是说，对于一个 OBF 中断，当表示“扫描码”端口为满的 `_obf = 0`，并且表明中断驱动数据传输的 `inte = 1` 时，`intr` 变为 1 (触发中断)。

当图 9-16 中的一个键按下时，键盘 DC 发送一个扫描码到键盘 DCI，这将写入扫描码到 DCI 中的“扫描码”寄存器，并设置 OBF 位 (`_obf = 0`)。当 `inte = 1` 时，这将触发中断请求信号 `intr`，使 `intr` 变为 1。`intr = 1` 将触发一个处理器中断服务程序。中断服务程序将读取“扫描码”寄存器中的扫描码，并存储到主存储器中。因为这是一个中断驱动传输，中断驱动程序不需要轮询 OBF 状态位；`intr = 1` 说明了“扫描码”缓冲区为满。当一个输入端口的读周期开始时 (见图 9-11)，`_rd` 的 1-0 转换将清除 INTR 位，使得 `intr = 0`。当处理器读取“扫描码”缓冲区中的扫描码时，`_rd` 的 0-1 转换将清除 OBF 位，使得 `_obf = 1`，表明该缓冲区现在为空，此时，DCI 可写另一个扫描码到“扫描码”缓冲区中。

9.5.2 程序控制传输

程序控制传输，也被称为程序控制 I/O，并不是中断驱动的；相反，处理器通过一个定时器触发，周期性地执行一个轮询程序，通过检测各个 DCI 中的 I/O 端口状态位来确定哪一个设备需要服务。轮询有一定的优先级次序。下面的程序说明了一个程序化传输，其检测了每个设备的 OBF 和 IBF 标志位。程序可在 `ibf = 1` (输入缓冲区为满) 时从端口输入数据，或在 `_obf = 1` (输出缓冲区为空) 时输出数据到端口。该程序代码是用于图 9-17 中的键盘端口 0，其他设备的程序代码部分标记为 DCI_X、DCI_Y 等。

395

实现程序控制 I/O 的轮询程序：

```

POLL:      ...          //start polling programmed I/O DCIs
...
DCI_KB:    LD (Port_1)    //input keyboard status
            AND 1         //select status bit 0, _obf;
            //bit 1 is ibf
            CMP 0         //is _obf = 0? 0 indicates full.
            JNE KIBF      //if output buffer not full, check
            //keyboard IBF
            LD (Port_0)    //else, output buffer full, read
            // "Scancode" port, assuming
            //memory-mapped I/O port
            ST ...        //store the scan code in memory
KIBF:      LD (Port_1)    //check keyboard IBF flag
            AND 2
            CMP 2
            JEQ DCI_X      //if input buffer not empty, check
            //the DCI of another device X
            ...           //else, input buffer empty, read data
            //from memory and write to "Data"
            //port
DCI_X:      ...

```

DCI_Y: ...
 ...
 ...

在这种情况下，处理器不仅参与到设备和存储器之间的实际数据传输，也直接涉及每个设备的周期性的轮询。只有当系统中有大量设备时，程序化传输才有较高的效率。例如，当使用计算机来监控一个工厂中的多个传感器时，如果使用中断驱动 I/O 机制，这将导致处理器经常性地切换到中断处理程序，会浪费可用的处理器时间。然而，可以通过一个主控制器接口，如 USB 主控制器接口来执行程序化传输，主控制器接口在需要服务时会中断处理器。

例如，思考一个现代个人计算机系统，其包含多种类型的 USB 外围设备，例如数字扬声器、数字电话、数字照相机、数字传真机、可移动 flash 存储器等，如图 9-18 所示。所列的 USB 接入设备可每天变化。一般的，一个 USB 主控制器接口可以轮询 127 个 USB 设备。

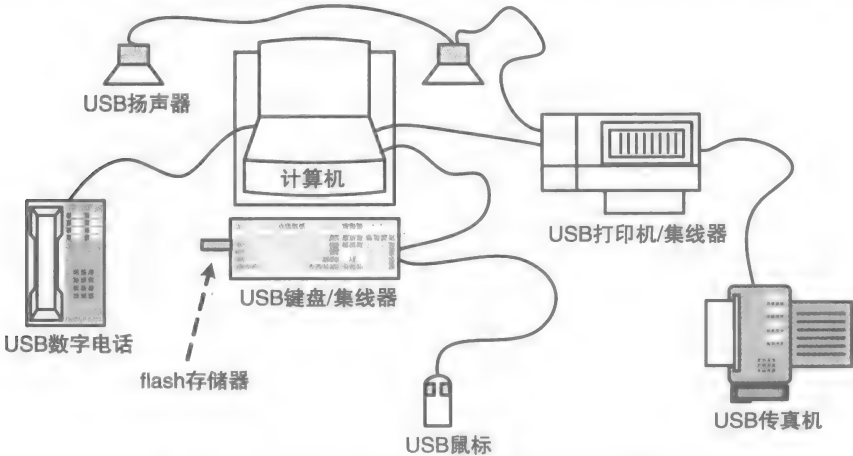


图 9-18 连接多个 USB 设备的现代个人计算机 [4]

USB 主控制器通过包来和各个设备进行通信，包是多个数据域的集合，每个数据域包含一定量的信息。图 9-19 说明了图 9-18 中的 USB 设备与一个包含一个根集线器和三个端口的 USB 主控制器接口的连接方式。如上所述，打印机中的 USB 集线器可用于连接 USB 扬声器和 USB 传真机，在键盘中的 USB 集线器可用于连接 flash 和鼠标。

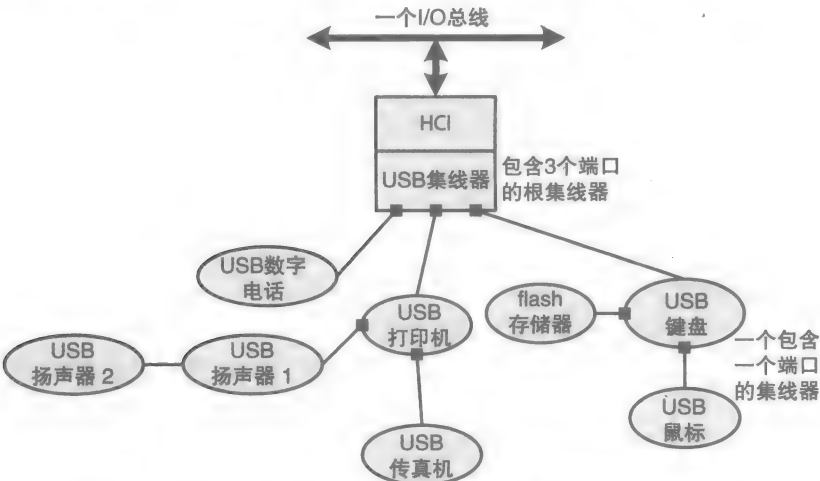


图 9-19 USB 设备连接到一个包含三个根集线器的 USB 主控制器

表 9-4 用于 USB 设备的 4 种包类型

| 包类型 | 描 述 |
|-----|---|
| 中断 | 这种包是用于与如键盘和鼠标等需要中断驱动传输的外围设备进行通信。取代向处理器的直接中断，设备将发送一个中断包给它的 HCI |
| 同步 | 这种包是用于与如数字音频、视频设备等需要与系统实时通信的外围设备进行通信。在这种设备中，数据传输速率比数据精度更重要 |
| 批量 | 这种包是用于与如打印机等需要传输批量数据的外围设备进行通信。在这种情况下，数据精度比数据传输速率更重要 |
| 控制 | 这种包是用来监控 USB 集线器并设置为设备连接或断开 |

所有连接到系统的 USB 设备请求的服务按其传输包的优先级分为 4 类，如表 9-4 所示。发送给各个 USB 设备的包会进一步封装成帧，从 USB 主控制器接口发出，依次通过 USB 根集线器、USB 集线器和 USB 端口、USB 电缆发送到各个 USB 设备。反之亦然，通过 USB 集线器和 USB 根集线器从 USB 电缆和 USB 端口发送到 USB 主控制器接口。

如果帧通信的频率足够高（每几毫秒），其可以捕获到如键盘和鼠标等请求中断驱动 I/O 的设备对即时性服务的延迟中断请求，也可以传输数据到 / 从诸如数字扬声器、数字电话等需要实时数据通信的外围设备。一个 bulk 包是用于与慢速设备如打印机进行通信，一个控制包是用于监测 USB 设备来确定设备是否连接。

当一个主控制器同时与多个设备进行通信时，与各个设备进行通信的基本操作还是相同的。每一个由主控制器发出到各个设备的包必须包含 I/O 端口地址和服务类型，服务类型为写端口和读端口。对于一个读端口的包，设备接收到该包后，会发送一个包含设备数据的（如扫描码）反馈包给主控制器，主控制器继而存储该数据到主存储器中。而对于一个写端口包，主控制器必须先访问主存储器，并从中获取设备数据，再发送该数据到设备中。

因为在主控制器和设备之间通信的数据大小是按字节数量进行排序的，包括主控制器和每个设备的 DCI 在内都包含一定的存储器作为缓冲区来临时存储接收到的或要发送的数据。关于 USB 主控制器接口的更详细描述将在 9.7 节介绍。

398

9.5.3 DMA 传输

中断驱动传输和程序化传输在设备和存储器之间的数据传输在数据量较小时（几字节）效率较高。而对于如磁盘驱动器等需要传输大量（如 4KB）数据到 / 从存储器的设备，采用不需要处理器参与到实际数据传输过程的 DMA 会更有效。为了更好地说明 DMA 传输，请参考图 9-20 的简单系统。

包括处理器和 DMA 控制器在内都需要访问存储器，一般情况下，需要一个被称为仲裁器的仲裁模块来决定哪个设备访问共享的总线。然而，在一个如图 9-20 所示的简单系统中，处理器也被用作仲裁模块。下面的步骤说明了一个 DMA 传输过程：

1) 操作系统使用处理器来写 DMA 控制器和 DCI 的 I/O 端口从而初始化 DMA 传输。处理器（通过设备驱动）向存储器地址寄存器（MAR）写入传输数据的存储器起始地址，并向字节计数寄存器（BCR）中写入要传输的数据字节数。处理器也向配置寄存器（CR）端口中写入表明数据传输方向（从 / 到存储器）和 DMA 传输类型，DMA 传输类型有连续的，例如当启动一个计算机时，或不连续的，例如在系统正常运转过程中，在 DMA 传输数据时处理器需要使用总线。处理器也会写数据到例如磁盘 DCI 等的 I/O 端口来配置磁盘进行一个读或写操作。一旦两个控制器都已经设置好，处理器将通过配置磁盘 DCI 中的“start”位来

开始一次 DMA 传输。

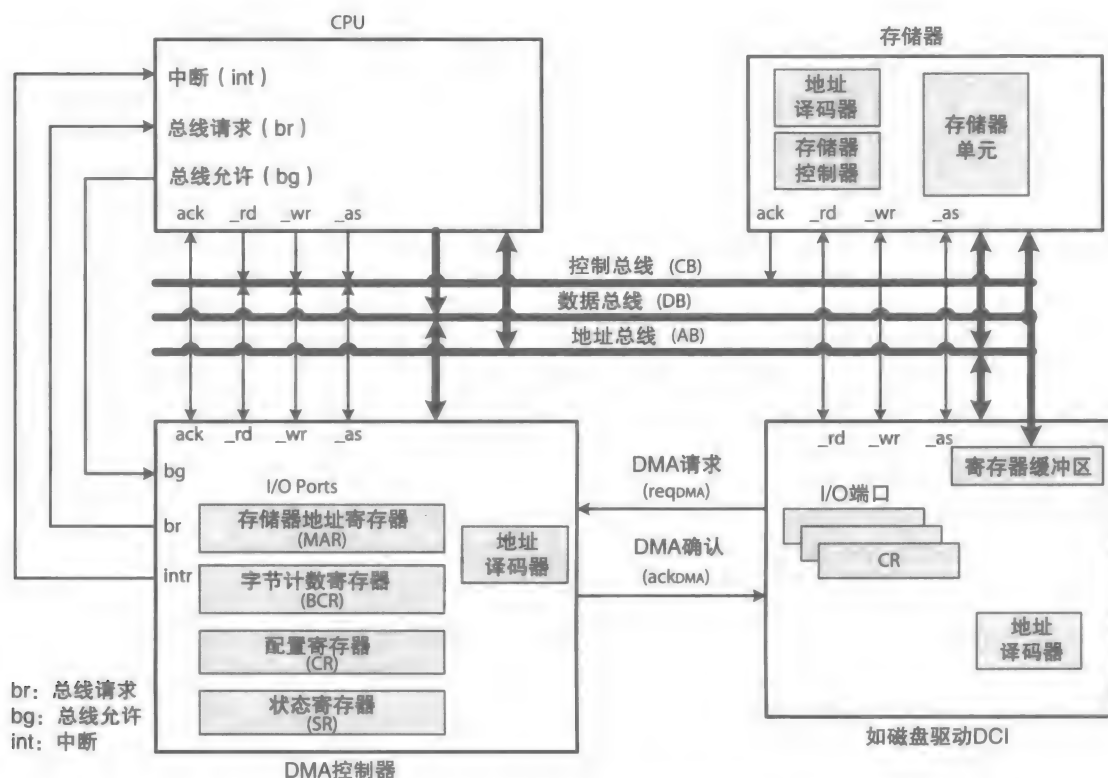


图 9-20 通过一个简单地 DMA 控制器来说明 DMA 传输，假定采用存储器映射 I/O 端口

2) 磁盘 DCI 通过与控制磁盘硬件的磁盘 DC 和 DMA 控制器进行通信从而完成从 / 到存储器的数据传输。假定 DMA 数据传输方向为从设备到存储器，DCI 首先存储设备数据到它内部存储器（参见例 9-1 中 Samsung 磁盘驱动器），接着选中 DMA 请求信号（req_{DMA}）请求一次 DMA 传输。继而 DMA 控制器将选中总线请求信号（br）请求处理器释放总线。处理器一旦完成当前的总线周期（如果有），将释放总线并选中总线允许信号（bg），总线允许信号用于允许 DMA 控制器使用总线。当处理器关闭所有与总线相连的三态缓冲器时，总线被释放。一旦 DMA 控制器接收到选中信号 bg，其变为总线主控并可以初始化存储器写周期。一个 DMA 读周期执行过程与之相似，不同之处在于数据从存储器传送到 DCI，然后从 DCI 传送到 DC。

3) 作为一个总线主控，DMA 控制器使用 MAR 中的内容作为下一个存储器地址来开始一个存储器写周期。它也为 DCI 选中 DMA 确认信号（ack_{DMA}）使之可放置数据到数据总线上。当进行一次总线传输后，MAR 将增加，而 BCR 将减少。对于一个存储器读周期，当存储器数据放置到总线上时（如图 9-7），ack_{DMA} 将选中，表示其准备好传输数据到磁盘 DCI。

4) 一旦 DMA 控制器完成一次传输，它将做以下两件事之一：当传输类型为连续时，它将传输下一个数据项（重复步骤 3），当传输类型为不连续时，它将释放总线，DMA 控制器解使能 br 信号，总线被释放。另一个 DMA 传输可以从第 2) 步开始，一旦所有的数据传输完毕，BCR 变为 0，DMA 控制器将中断处理器，告知本次 DMA 传输已完毕，若当前还需要另一次 DMA 传输，处理器可对其初始化。

一个现代 DMA 控制器，然而，可以通过实现多个 DMA 通道来给多个设备提供服务，每

个 DMA 通道配有可配置的 I/O 端口从而来服务不同的设备。一个现代的 DMA 控制器也可以包含端口使其与处理器的通信最小化，从而提升性能。例如，处理器通过在存储器中生成一个 DMA 传输表（或链表）并传输该表的地址给 DMA 控制器，从而在每次数据传输时不需要处理器来初始化 DMA。表中每一项都包含了初始化一个独立的 DMA 传输的必要信息。当一次 DMA 传输完成时，DMA 控制器将自动获取表中下一条信息从而在不中断处理器的情况下初始化下一次 DMA 传输。DMA 控制器只会在完成整个表项的数据传输后才会向处理器发出中断。

400

对于某些现代 DMA 控制器，处理器也可以配置 DMA 控制器使其每当完成数次 DMA 传输时向处理器发出中断。在这种情况下，处理器可以检查 DMA 的状态并更新传输表，例如，如果需要，可向该表添加更多的 DMA 传输。

当一个现代多通道 DMA 控制器被用于在两个存储器单元间传输数据时（如存储器 - 存储器 DMA）也需要以不同的方式运作。从一个单元的存储器读周期后可能紧跟着一个向另一个存储器单元的存储器写周期。最后，每个设备也可以包含其专用的 DMA 控制器来代替共享的多通道 DMA 控制器。在这种情况下，多个 DMA 控制器将通过竞争来与主存储器通信。

如我们将在第 9.8 节看到的那样，一个 USB 主控制器接口包含两个 DMA 控制器。主控制器使用一个 DMA 控制器来完成主接口和主存储器之间的数据传输，使用另一个控制器来完成主接口和其他连接的设备的数据传输。

9.6 中断

典型的，中断分为**硬件中断**和**软件中断**，硬件中断是由内部的或外部的硬件模块向处理器发出的中断，软件中断是通过执行一条特殊的指令，如“INT”，来调用一个系统级的程序。某些内部硬件中断被称为**同步中断**、**异常**或**陷阱**，这些中断的产生是由发生在 CPU 数据通路内的一些诸如算术溢出、除以零、非法操作码等错误造成的。因为这些中断是指令相关的，所以被称为陷阱。如果一个程序执行结果异常，例如，在一个特殊的算术指令——如在存储器地址 X 的“ADD”指令——有算术溢出，无论程序执行多少次都将在执行该指令时溢出（假定算术溢出的中断是启用的并且程序的输入是相同的）。

然而，其他的内部硬件中断可能是不同步的。例如，考虑一个采用**多道程序设计的执行环境**，其中程序经常因为太大而无法填充到主存储器中。在执行一个程序时目标指令或数据并不在主存储器中时，将导致一个中断，一般被称为**缺页中断**。在程序可被唤醒之前，需要使用一个 DMA 来从磁盘驱动器传输所需求的程序代码或数据页（如 4KB）到主存储器中。随着页在不同的程序中移入移出存储器，发生缺页中断的时间可能不同步。缺页中断将在第 10 章中进一步讨论。

[在一个采用多道程序设计的执行环境中，任何种类的中断都将停止当前运行程序（被称为进程）的执行。在发生一个缺页中断的情况下，操作系统将初始化一个磁盘 - 存储器的 DMA 传输，并放置中断进程的 ID（如进程 - 1）到一个**等待队列**中。当该页正在传输时，操作系统将唤醒或开始另一个进程的执行。当这次 DMA 传输完成时，DMA 控制器将中断当前执行的进程（如进程 - 2），控制权交还给 OS，OS 之后将进程 - 1 的 ID 从等待队列移动到被一个准备执行队列，这个准备执行队列被称为**准备队列**。

401

在准备队列中的进程（包括进程 - 1）将轮转使用一部分处理器时间，这个时间叫作**时间片**。使用一个定时器模块来控制时间片的持续时间，当前时间片耗尽时，定时器模块将发生一个中断。每个进程可使用一个或多个时间片来完成执行过程。一旦发生一个时间片中断，OS 将放置被中断的进程的 ID 到准备队列中并将处理器使用权限交给准备队列队首 ID

对应的进程。当轮转到进程 - 1 (和准备队列中的其他进程相似) 时, 进程 - 1 将恢复执行, 直到发生另一个中断, 中断包括另一个缺页中断或时间片中断等。]

硬件中断, 例如 DMA 中断和 DCI 中断, 并不是处理器的内部中断, 被称为**异步中断**。这些中断可以在任意程序执行过程中的任意时间任意地点发生。例如, 当一个系统使用中断驱动 I/O 来给鼠标和键盘提供服务, 鼠标移动和键按下的准确时间是不可知的。

Intel 的“INT”和 ARM 的“SWI”是软件中断指令的示例。软件中断总是同步的, 其可以用于多种目的, 包含允许程序轮转和访问共享资源, 共享资源如可读写文件的磁盘驱动器。这里, 我们使用术语中断来表示所有类型的中断。

9.6.1 中断处理

图 9-21 说明了调用硬件中断处理程序 (IH) 和子中断调用 (例如函数、程序或方法) 之间的区别。当一个程序执行跳转 / 分支子程序指令时, 如包含子程序起始地址的“JSUB sort”, 子程序将被调用。在图中, 子程序或 IH 的起始和结束地址被标记为 B 和 C。跳转指令开始了一个子程序的执行, 如图 9-21a 所示, 从 A 到 B 的实线箭头阐述了调用过程, 而从 C 到 D 的箭头为子程序的返回。在执行一个子程序之后, 控制权交回给调用程序, 返回到程序唤醒执行的地址 D。

另一方面, 一个硬件中断处理程序的执行并不是通过一条指令来开始的, 通过图 9-21b 中的虚线箭头来表示。在这种情况下, CPU 必须首先发现产生中断的原因, 并据此决定调用哪一个 IH。例如, 如果产生中断的原因是鼠标移动, 鼠标 IH 将被调用。

当前指令执行完毕后, CPU 将检测为中断等待的其他请求, 例如检测图 9-17 中 intr 信号是否被选中。例如, 如果 CPU 发现与鼠标相关的 intr 信号被选中, CPU 将调用鼠标 IH。
[402] 鼠标 IH 将与鼠标 DCI 进行通信, 访问其端口并读入鼠标位移信息, IH 将根据位移信息移动屏幕中的鼠标。

软件中断处理程序一般是用整数来标号的, 例如, 奔腾的指令“INT 3”, 表示一个断点异常, 用于调试工具, 以便在测试时为程序创建一个断点 [5]。当程序执行到断点时, 当前主程序将被中断, 调用一个中断处理程序。

当调用 IH 或调用子程序时, CPU 的状态, 例如, 图 8-7 (第 8 章) 中 ACC、X、SR 和程序指针 (PP) 的内容, 将被 IH 或子程序保存在 CPU 内部或在主存储器中。当 IH 或子程序执行结束时, 控制权将返回给被中断的或调用的程序, 并恢复保存的状态。保存的 PP 叫作返回地址或返回程序指针 (RPP)。在图中, 子程序的 RPP 是地址 D, 而 IH 的 RPP 会是地址 A 或地址 D, 具体将在下面讨论。

1. 确定中断

确定中断指的是根据产生中断的原因, 由 CPU 执行的一系列规则。下面是应用于确定中断的一系列规则:

1) 所有在 RPP 之前的指令必须已执行完成并且 CPU 状态恢复完毕。

2) 所有从 RPP 开始的指令均未开始执行。

3) 若中断是由同步事件产生的, 除了软件中断, RPP 必须指向导致本次异常的指令。此外, 如果本次中断是由执行单元提出的, 例如算术溢出, 执行的结果不会改变处理器的状态。例如, 图 8-7 (第 8 章) 中当“ADD”指令造成算术溢出时, ACC 寄存器的内容不会改变。ACC 中的内容会被相应地 IH 分析处理。

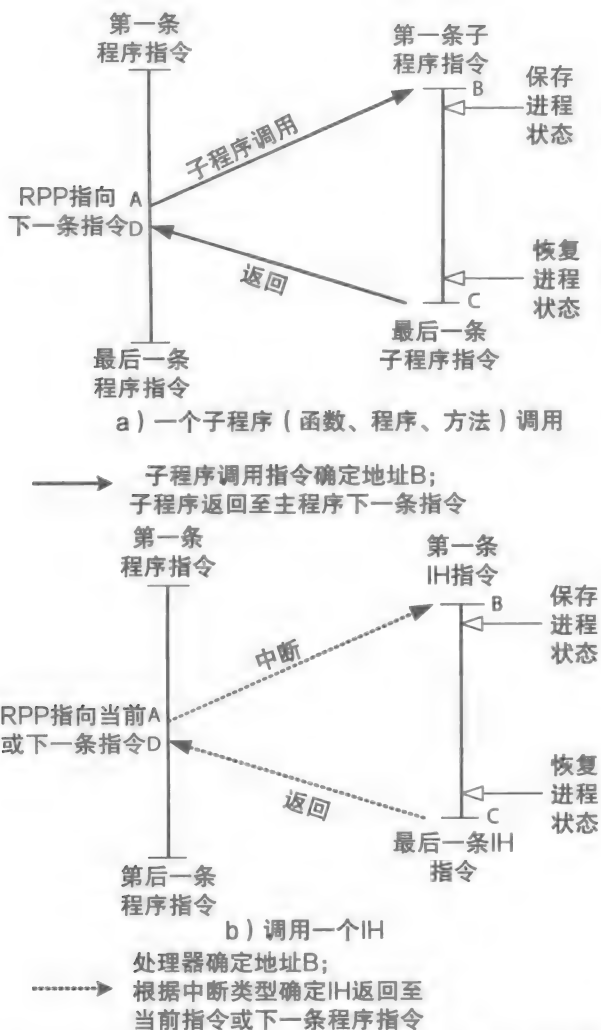


图 9-21 子程序调用和中断处理程序调用；虚箭头与实箭头

4) 如果是异步(外部的)事件引发的中断或软件中断, RPP 必须指向下一条指令。

如果 CPU 的数据通路是单周期的, 在执行当前指令时执行中断检查, 如 9.7 节所述。如果 CPU 数据通路是多周期的, 将在退出指令的最后一次数据通路操作期间执行中断检查。如果 CPU 数据通路是流水线的, 中断检测将在写回阶段进行, 写回阶段表明当前指令执行完成即将退出。

在流水线 CPU 中的确定中断比在单周期或多周期数据通路中更加复杂。因为在流水线中多个阶段同时运转, 可能在同一时间产生多条中断请求。在这种情况下, 指令地址(例如 PP 中的内容)将伴随着已经被取的指令从取指令阶段转移到译码阶段。指令地址和在每个阶段生成的任意中断请求都将从该阶段转移到下一个阶段直到转移至写回阶段。

当发生一个中断时, RPP 作为保存在主存储器中的 CPU 状态的一部分保存在一个内部的寄存器中, PP 的内容将被改变转而执行一个 IH。如果 CPU 的数据通路采用流水线, 这也将造成一次流水线清空。注意在寄存器中的内容不会被清除而是保留下来使得其可以被 IH 存储到主存储器中。正如我们早先讨论的那样, CPU 的状态也可以保存在 CPU 内部(见练习部分)。关于确定中断在流水线 CPU 中更详细的介绍可参考其他书籍。

2. 向量中断

向量中断指的是系统中存在多个 IH，另一方面，非向量中断意味着在其所有中断处理中只有一个 IH。图 9-22 分别说明了存储器中一个单独的 IH 和多个 IH 的组织结构。在触发非向量中断的情况下，每次发生中断时只会触发一个单独的 IH，中断处理程序之后将对优先级最高的中断设备提供服务。非向量中断机制实现方式比较简单，但是因为所有的中断都是由一个例程来处理的，中断无法迅速被处理，因此在现代计算机系统中非向量中断并不常用。然而，如将在 9.7 节讨论的，非向量中断机制可用于简化说明 CPU 中断处理的数据通路细节。

下面列出了实现向量中断的具体要求：

- 必须在主存储器中保留一小块区域以存储中断向量表，中断向量表的起始地址标记为“基地址”，如图 9-22b 所示。
- 从每个 DCI 和 DMA 控制器中发出的中断请求必须先在硬件中进行优先级排序从而生成一个独特的数字，这个数字被称为中断向量 (IV)，其与最高权限的 IH 相关。IV 是用于调用一个相应的中断处理程序。
- 在系统启动过程中，随着跳转指令的执行，中断向量表中的每个条目必须填入不同的 IH，每个 IV 对应一个 IH。
- 当处理器采用向量中断时，处理器的数据通路必须通过添加新的寄存器和电路的扩展方式来执行新的指令和特性。

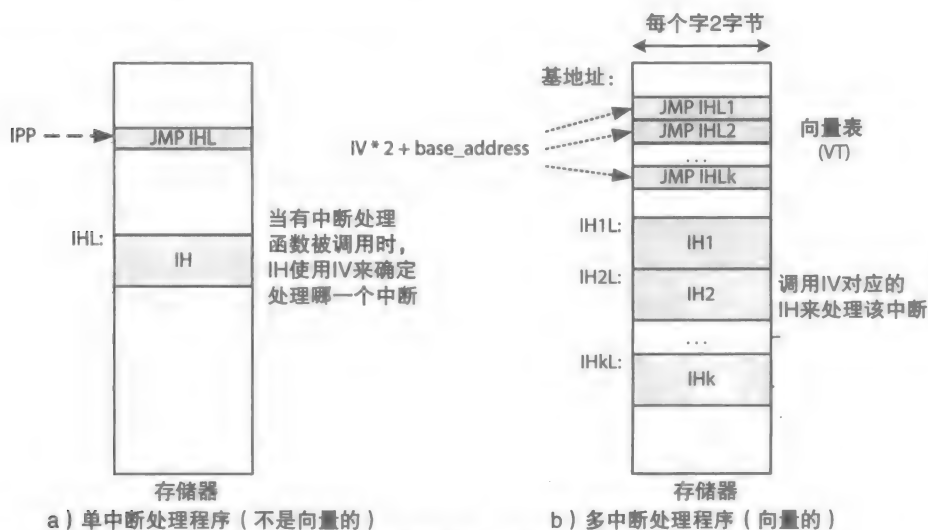


图 9-22 向量中断的处理函数；假设存储器中每个字 2 字节

公式 (9-2) 说明了如何通过一个 IV 和中断向量表的 base_address 的线性函数来确定中断处理指针 (IHP) 的示例。如 $IV = 1, 2, 3$ 等, $IHP = \text{base_address} + 2, \text{base_address} + 4$ 等, 其是在假定每个条目为 2B 时访问中断向量表中某个条目的索引。IV = 0, 如后文所述, 可以用于表示没有任何中断请求。

$$IHP = IV * 2 + \text{base_address} \quad (9-2)$$

图 9-23 说明了多个已进行优先级排序的 IH 的执行顺序。具有更高优先级的中断处理程序总是可以中断具有低优先级的程序, 但是反过来不可以。此外, 当调用中断处理程序时,

包括中断程序的返回地址在内的处理器状态必须被保存（一般保存在存储器中）。

下列描述了 CPU 调用一个 IH 的相关步骤；假定了 IV 已经进行了优先级排序，IH1（IV = 1）相对于 IH2（IV = 2）具有较低的优先级，而 IH2 对于 IH3（IV = 3）具有较低的优先级，等等，如图 9-23 所示。此外，IV = 0 是用于表示没有任何中断请求：

1) 当处理器接收到一个中断 IV 时（标记为 IV_r），如果当前 CPU 无法检查该中断请求的特性或者当前执行的 IH（标记为 IV_c）优先级比 IV_r 高，处理器将暂时忽略该中断请求。这也就是说，当 IV_r ≤ IV_c 时，CPU 将忽略 IV_r，保持该中断请求挂起，直到当前的与 IV_c 相应的中断处理程序结束。另外，如果 CPU 可检查中断请求特性并且 IV_r > IV_c，CPU 将保存当前执行的 IH 的 RPP 和 IV_c 到 CPU 中的特殊寄存器中，并使用 IHP = IV * 2 + base_address 改变 PP 的内容，改变 IV_c 的优先级，使得 IV_c = IV_r。例如，当 IV_c = 1 并且 IV_r = 2 时，CPU 将为 IH1 保存 RPP1，并保存 IV_c = 1，使用 base_address + 4 代替 PP 中的内容，并改变 IV_c = IV₂。

2) CPU 在存储器地址 IHP 执行跳转指令，开始执行与当前 IV_c 对应的 IH（如当 IV_c = 2 时为 IH₂）。

3) 调用中断处理程序之前，必须先保存处理器状态，保存包括 RPP、IV_c 等到存储器中（见 9.6.2 节）。然而，CPU 将继续监视 IV_r 以应对更高优先级的 IV。

4) IH 将对如 DCI 或 DMA 控制器等中断设备提供服务。一旦完成，中断处理程序将恢复包括保存的 RPP、IV_c 等处理器状态，返回到被中断的程序，使得其可以被唤醒执行，被中断的程序可以是较低优先级的 IH（如 IH₁）或系统程序或应用程序等。

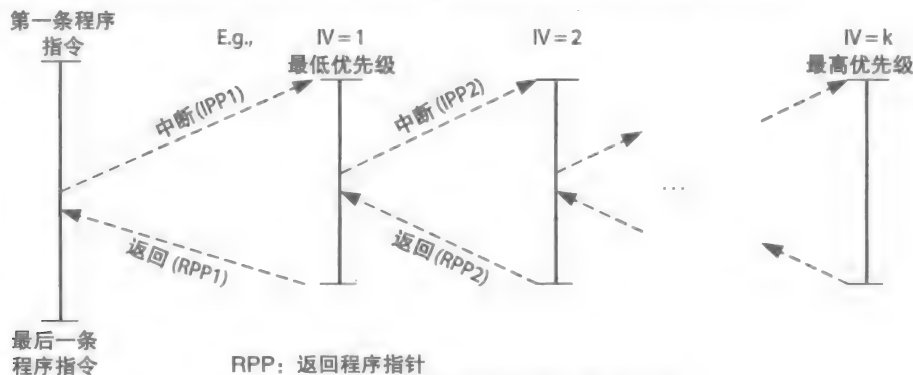


图 9-23 多个中断处理程序的调用过程

9.6.2 中断结构

图 9-24 说明了对两种中断请求进行优先级排序的硬件中断结构，菊花链式结构和独立请求结构。这是用于确定中断设备的 IV，IV 继而用于确定调用哪个 IH 来为中断设备提供服务。这些结构的细节将在下文说明。

1. 菊花链

图 9-24a 中的结构将所有中断请求设置为固定的优先级。当信号 int 被选中时，CPU 将选中一个中断确认信号 (iack) 来确定最高优先级的 IV。

关于菊花链模块的详细电路模型如图 9-24b 所示。当 IV_r 被选中时，iack 信号将从一个菊花链模块转移到另一个模块，直到遇到最高优先级（在那一时刻）的模块。在设置过程中

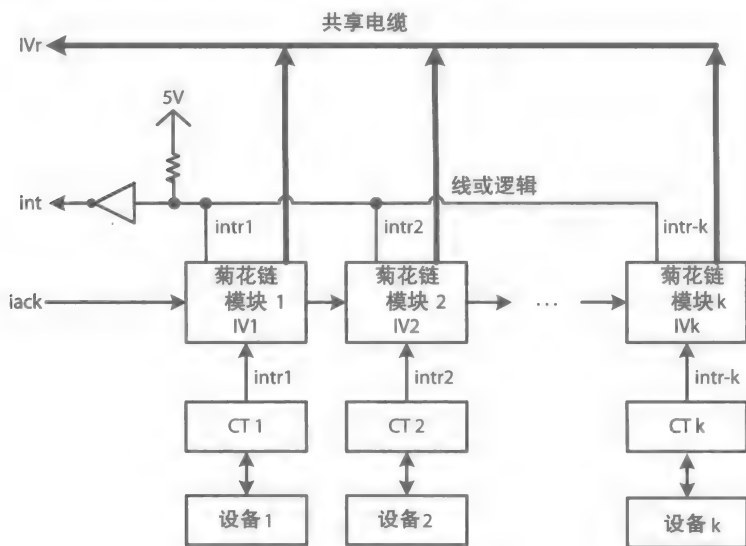
每个 DCI 将被分配一个独特的 IV。对于 USB 设备，只有 USB 主控制器接口可中断处理器。

虽然图 9-24a 中的结构是可扩展的并且更多设备可添加到系统中，其劣势在于位于菊花链尾部的 DCI 可能会产生饥饿现象。当到达位于菊花链尾部具有最低优先级的模块时，具有较高优先级的 DCI 可以阻止 $iack = 1$ 。

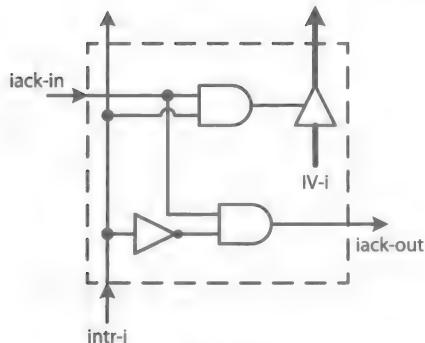
2. 独立请求

图 9-24c 中的结构是使用一个优先级编码器来快速确定最高优先级的 IV。在图中，8-3 优先级编码器通过连接输入信号 0 到地上实现一个 7-3 编码器。在这种情况下，当 7 个中断请求信号都未激活时， $IV_r = 0$ 。此外， $IV_r = 0$ 用于表示当前处理器无任何挂起的中断请求。这将消除图 9-24a 中的 int 信号，减少一个 CPU 输入信号。

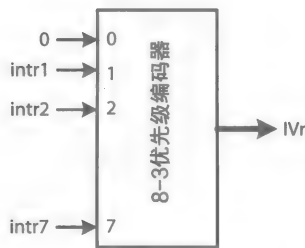
除了使用混合结构——部分独立请求结构，部分菊花链式结构——来实现中断优先级，独立请求结构都不需要 $iack$ 信号。在一个系统中，所有的 DCI 都将被分类为多个不同的优先级类。例如，所有的磁盘驱动器（如硬盘和 CD 盘驱动器）可以被视为设备类的一部分。在这种情况下， IV_r 用于分辨同一个类中的所有 DCI（如所有的磁盘驱动器）， $iack$ 信号将用于分辨 IV_r 对应的类中某个特殊的设备（如硬盘驱动器）。



a) 菊花链式结构



b) 菊花链模块



c) 独立请求结构

图 9-24 中断结构：a) 菊花链式结构；b) 菊花链模块；c) 独立请求结构

9.7 设计示例：中断处理 CPU

这一小节说明了中断处理 CPU 的数据通路需要包含以下要求和特性：

- 图 8-7（第 8 章）中的单周期累加器 ISA CPU 的数据通路是可扩展来实现给多个设备提供服务的单一 IH 的调用机制。
- IHP 是硬编码到 CPU 内的存储器地址 0x40，假定地址宽度为 8 位。在这里，地址 0x40 是任意选择的。
- IH 可最多服务 16 个设备，包括一个定时器模块、一个 DMA 控制器和 1 ~ 14 个 I/O 设备。
- 菊花链式结构（图 9-29a）是用于给从 16 个设备发出的中断请求进行优先级排序，其中 IV = 15（最高权限）是分配给定时器模块，IV = 14 分配给 DMA 控制器，0（最低权限）分配给键盘。注意，应为我们使用菊花链式中断结构，我们也可以使用 IV = 0 作为一个合法的设备 IV。
- 一旦发生一个中断，CPU 的状态由 ACC、X、SR 等寄存器的内容定义，将被 IH 保存在存储器中。返回地址（RPP）将被保存在 CPU 内部。

图 9-25 阐明了一个 16 级菊花链式中断结构，其对包括一个定时器、一个 DMA 和 1 ~ 14 个 I/O 中断请求（图中只显示了一个键盘）在内的设备进行优先级排序。每个设备的 DCI 将生成一个中断请求（intr）信号，其在设备需要 CPU 提供服务时将变为 1。例如，当键盘中某个键按下时将设置其中断请求信号 intr0 为 1（也可参见图 9-17），使得 CPU 中断当前执行的指令并开始执行服务键盘的 IH。

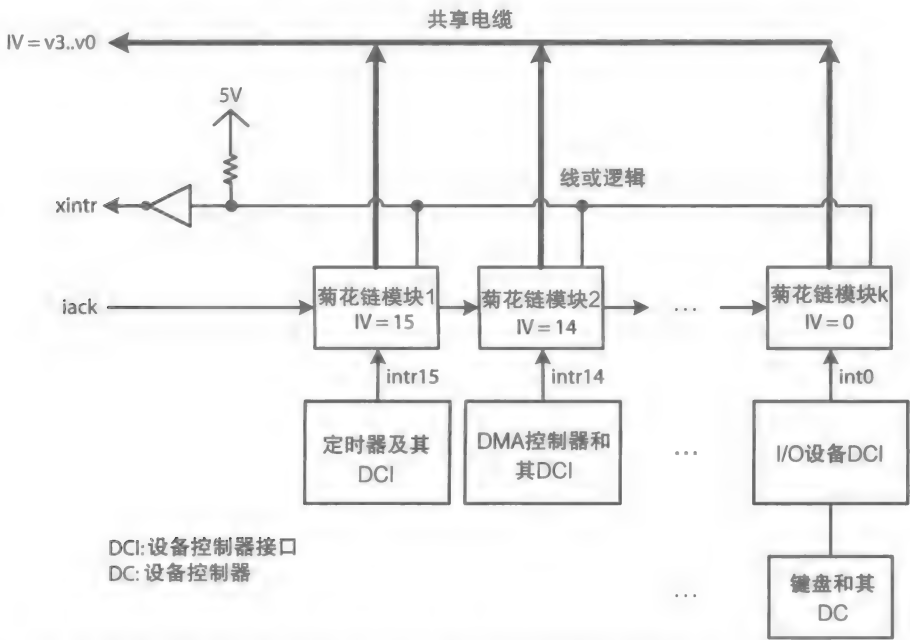


图 9-25 一个 16 级菊花链式中断结构，对 16 个设备的中断请求按最高（IV = 15）到最低（IV = 0）进行优先级排序

相似的，当 DMA 完成一次 DMA 传输时，DMA 控制器将选中其中断请求信号 intr14（也

可参见图 9-20)。定时器模块也可以向 CPU 发出中断, 选中其中断请求信号使 $\text{Intr15} = 1$, 因此 OS 可以启动或唤醒另一个程序的执行。一个 I/O DCI 可以是专用的也可以是通用的, 例如 USB 主控制器接口。

从定时器模块、DMA 控制器和最多 14 个 DCI 中发出的 intr 信号通过线或逻辑产生一个外部中断请求信号 xintr , 如图 9-25 所示。 xintr 信号与 CPU 连接, 当 CPU 中断处理特性启用并且 xintr 信号选中时, 将产生一次中断。中断处理 CPU 的数据通路如图 9-26 所示。

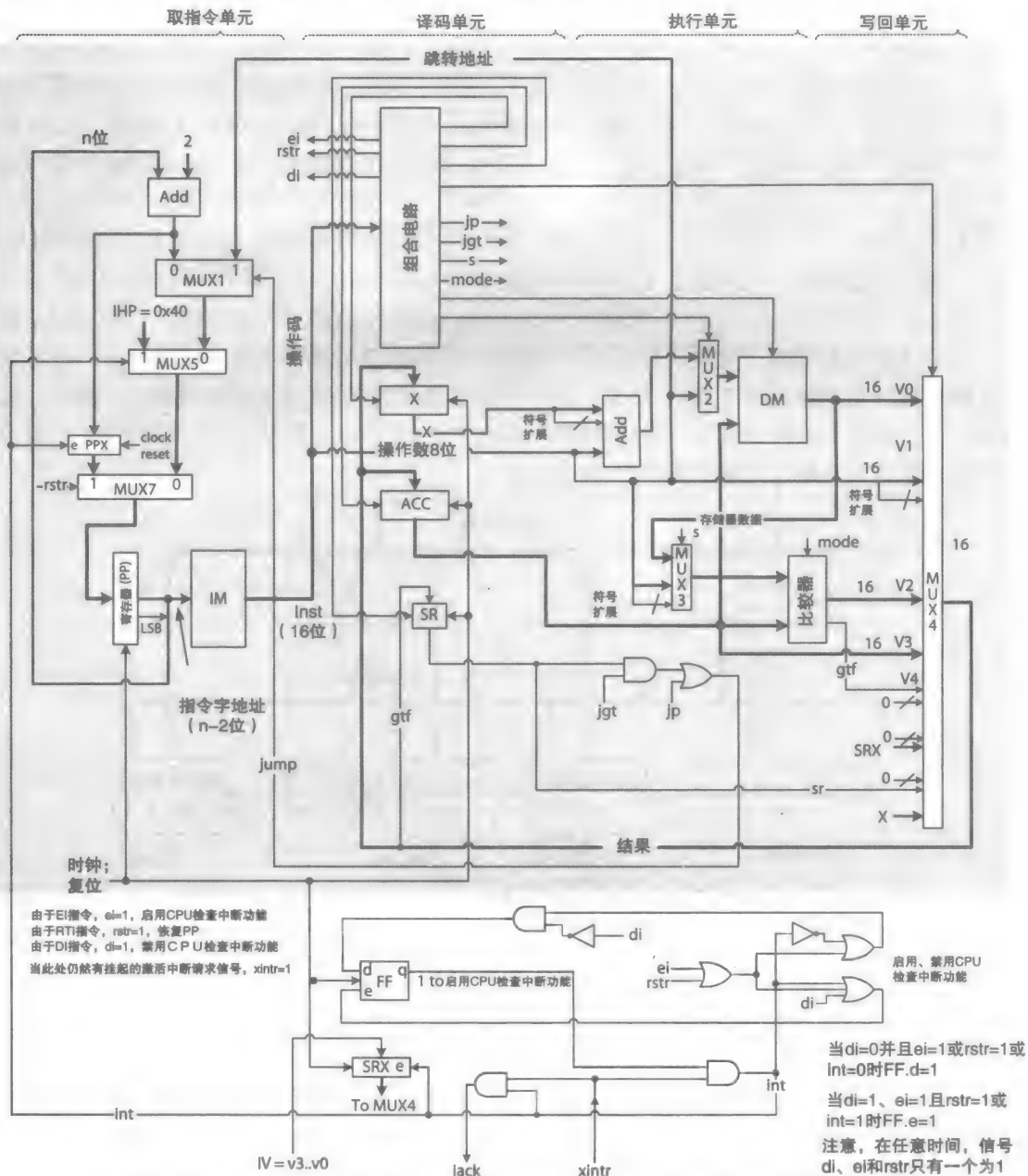


图 9-26 中断处理 CPU 的数据通路，其是对第 8 章中的累加器 ISA 的扩展

新的数据通路实现了 8 条新指令，如表 9-5 所示，其又包含了下列新加的硬件：

- 一个 D 触发器 (FF) 是用于使能或不使能 CPU 检查中断特性。当 “EI” 或 “RTI” 指令执行时, FF 将会被设置为 1 (例如 $FF.q = 1$)。
- 添加了一个辅助的程序指针寄存器 (PPX) 用于保存 RPP。在这种情况下, RPP 中存储的是当前执行程序的下一条指令的地址。当从中断处理程序中返回时, 程序将从保存在 PPX 中的地址开始重新执行。当指令 RTI 执行时, 位于取指令单元的 2-1 MUX7 将选择 PPX (PPX 中的内容) 作为下一个 PP (PP 的内容)。
- 在取指令单元新加一个 2-1 MUX5, 在发生中断时其将选择硬编码的 $IHP = 0x40$ 作为下一个 PP。
- 在译码单元新加一个辅助状态寄存器 (SRX) 来保存具有最高优先级的中断设备对应的 IV。
- 指令译码单元可被修改, 其包含了 3 个或更多的信号, 为 ei (启用 CPU 检查中断功能)、rstr (恢复) 和 di (禁用 CPU 检查中断功能)。当 EI 指令执行时, ei 信号将被选中。ei 信号在下一个时钟周期设置 FF ($FF.q = 1$), 其将启用 CPU 检查中断功能。rstr 信号在 RTI 指令执行时也将被设置 FF ($FF.q = 1$)。当 DI 指令执行时, di 信号将被选中, 其将同步地复位 FF ($FF.q = 0$) 并禁用 CPU 检查中断的功能。
- 写回阶段的 MUX4 被一个 8-1 MUX 代替, 从而实现 $ACC \leftarrow SRX$ 、 $ACC \leftarrow SR$ 和 $ACC \leftarrow X$, 其将被 IH 调用保存 CPU 状态到存储器中。

一旦复位, FF 将被置为 0 ($FF.q = 0$), 禁用 CPU 检查中断的功能。FF.q 将一直保持为 0 直到系统启动并且操作系统完成初始化整个系统的以下动作:

- 加载单独的 IH 到存储器起始地址, 例如, 地址 $0x42$ 。
- 存储指令 “JMP $0x42$ ” 到 $IHP = 0x40$ 位置。
- 通过执行 EI 指令, 使得 $FF.q = 1$, 启用 CPU 检查中断功能, 开中断。

410
2
411

表 9-5 用于实现单中断处理机制的新指令

| 指 令 | 返 回 | 描 述 |
|---------------|--|--|
| DI (optional) | $FF.q \leftarrow 0$; | 关中断: 该指令是用于同步复位 FF ($FF.q = 0$)。当 FF 复位时, 当前执行的指令不可被中断 |
| EI | $FF.q \leftarrow 1$; | 开中断: 该指令用于同步设置 FF ($FF.q = 1$)。当 FF 被设置后, 如果 $xint = 1$, 当前执行的指令将被中断 |
| MVSR2ACC | $ACC \leftarrow SR$; | 移动 SR 中内容到 ACC: 该指令将复制 SR 寄存器中的 1 位数据 (用 0 填补) 到 ACC 寄存器中, 使得 SR 寄存器中的内容可被中断处理程序 (IH) 保存在存储器中 |
| MVX2ACC | $ACC \leftarrow X$; | 移动 X 到 ACC: 该指令复制 X 寄存器中的内容到 ACC 中, 使得 X 寄存器中的内容可以被 IH 保存到存储器中 |
| MVACC2SR | $SR \leftarrow \text{LSB of } ACC$; | 移动 ACC 到 SR: 该指令复制 ACC 的最低有效位 (LSB) 到 SR 寄存器中。IH 在返回时执行该指令来恢复被中断的程序的状况 |
| MVACC2X | $X \leftarrow ACC$; | 移动 ACC 到 X: 该指令复制 ACC 中的内容到 X 寄存器中, IH 在返回时执行该指令来恢复被中断的程序的状况 |
| MVSRX2ACC | $ACC \leftarrow SRX$; | 移动 SRX 到 ACC: 该指令复制 SRX 中的内容到 ACC 中, IH 执行该指令来确定中断设备的 IV |
| RTI | $FF.q \leftarrow 1, PP \leftarrow PPX$; | 从中断处理程序中返回: 该指令是 IH 中的最后一条指令。其将开中断, 同步设置 FF ($FF.q = 1$), 并返回使得被中断的程序可以被唤醒执行。注意, 某些程序在从异常中返回时调用该指令 |

系统正常运行过程中，可以有一个或多个中断请求信号 (intr) (如图 9-25 所示) 被选中。每当 xintr 信号变为 1 并且 CPU 开启检查中断功能时 (如 FF.q = 1)，在 CPU 内部的 int 信号变为 1，并导致在下一个时钟周期进行以下操作：

- 通过 MUX7, int = 1 将导致数据通路执行 $PPX \leftarrow PP + 2$ 。将 PP + 2 的值作为 RPP 保存起来。当从中断处理程序中返回后，被中断的程序将从 RPP 的指令地址唤醒执行 (见图 9-21b)。
- 通过 MUX5 和 MUX7, int = 1 将导致数据通知执行 $PP \leftarrow 0x40$ 。导致接下来执行位于存储器地址 0x40 的指令 “JMP 0x42”，从而开始单一 IH 的执行。
- int = 1 也将导致 $SRX \leftarrow IV$ 从而保存最高优先级设备的 IV。注意，当 int = 1 并且 xintr = 1 时，iack 将变为 1，反过来选择请求服务的最高优先级设备 (图 9-25) 的 IV。
- int = 1 也同步复位 FF，使得 FF.q = 0，这将关闭 CPU 检查中断的功能。当 FF.q = 0 时，int 将保持为 0，并阻止当前程序的执行，当前程序可能是一个在系统初始化过程中的一个 OS 例程，或是正常运行过程中的一个 IH。

例 9-2 中的伪码概括了单一 IH 在多个设备间为一个请求中断的设备提供服务的必要步骤。中断处理程序执行以下 4 个主要任务：

1) IH 保存被中断程序的状态到存储器中，被中断程序的状态指的是 ACC、X 和 SR 等寄存器中的内容。在伪码中，这部分在函数 save_cpu_status() 中，该函数包含以下代码：

```
STA Temp1;           // save ACC, for "STA" refer to Chap. 8
MVX2ACC
STA Temp2           // save X
MVSr2ACC
STA Temp3           // save SR
```

2) IH 必须确定中断设备的 IV，这样 IH 才可以调用一个驱动例程来给设备提供服务，并复位设备中断请求信号 (intr) ——参考 9.5.1 节键盘示例。这部分在伪码中是 “iv = get_iv()” 和 “switch” 语句。函数 get_iv() 将只用 MVSRX2ACC 指令来复制 SRX 中的内容到 ACC 中，接着将其中内容与 0、1、2 等比较，来确定中断设备的 IV。

3) 中断处理函数在返回之前必须恢复被中断程序的状态。这部分在函数 restore_cpu_status() 中，包含以下代码：

```
LDA Temp3;           // for "LDA" refer to Chap. 8
MVACC2SR             // restore SR
LDA Temp2
MVACC2X              // restore X
LDA Temp1            // restore ACC
```

4) IH 将执行 RTI 指令，导致数据通路在下一个时钟周期执行 $PP \leftarrow PPX$ 和 $FF.q \leftarrow 1$ 。这将恢复返回地址，恢复被中断程序的执行，并开始 CPU 检查中断功能。

例 9-2 通过伪码概述单一 IH 给如定时模块、DMA 控制器或 I/O 设备等中断设备提供服务的过程，每次给一个设备提供服务：

```
interrupt_handler_routine() //invoked when PP is set to
                           //interrupt handler pointer (IHP =
                           //0x40 in Fig. 9.26)
{
    save_cpu_status();      //save contents of ACC, X, and
                           //SR to memory

    iv = get_iv();          //get IV from the SRX
```



```
switch(iv)

15: handle_timer_interrupt(); break;          //IV = 15, highest
14: handle_DMA_interrupt(); break;

.
.
0: handle_keyboard_interrupt();               //lowest
endswitch;
restore_cpu_status();                         //restore contents of ACC, X,
                                              //and SR from memory
return_from_intrrupt();                       //RTI
}
```

当从一个中断中返回时，CPU 检测到 `xintr = 1`，中断处理程序将再次被调用来为另一个挂起 `intr = 1` 的设备提供服务。

最后，包含了表 9-5 中的关中断指令（DI）以防止使用软复位——例如，使用菜单中的“restart”选项。单一 IH 机制的劣势在于当 CPU 执行一个 IH 时，如果有另一个更高优先级的设备请求服务，更高优先级的设备必须等到当前中断处理程序完成其任务并返回时才能得到服务。因为这个原因，现代计算机系统实行向量化的中断机制使得 CPU 执行具有低优先级设备的 IH 时可被中断，从而执行高优先级设备的 IH（例如 DMA），如图 9-23 所示。

9.8 USB 主控制器接口

我们已经在 9.5.2 节中简单地讨论了主控制器接口的需求。设计 USB 主控制器接口的主要目的是将 CPU 从直接执行任务来为多个潜在的 I/O 设备提供服务这一繁琐过程中解放出来。在这种情况下，因为有太多设备发出中断，中断驱动传输机制可能并不实用，而程序化传输也不是一个可行性方案，因为其浪费了太多有价值的 CPU 时间。此外，USB 主控制器接口实行“即插即用”，从而在不安装设备驱动并系统重启的情况下支持多个 USB 设备。

414

9.8.1 标准

根据通用主控制器接口（UHCI）或开放式主控制器接口（OHCI）对传输速率为 1.5Mbps 的低速设备和 12Mbps 的全速设备的适用特性，设计了 USB 1.x 标准。另一方面，USB 2.0 标准是基于增强型主控制器接口（EHCI），设计可支持传输速率最高可达 480Mbps 的高速设备。USB 3.0 设计用于支持传输速率高达 5Gbps 的高速设备。每一代 USB 主控制器接口也包含根集线器来为更低速率的设备提供服务。例如，USB 2.0 主控制接口包含根集线器来服务低速设备和全速设备，此外，其还为高速设备提供服务。

一个 USB 电缆包含 4 个电线，其中有两个是电源和地，另外两个是数据信号线，标记为 D+ 和 D-。一个 USB 通信模块使用 D+ 和 D- 信号线来发送 / 接收采用 NRZI 编码方式的数据包（参考第 5 章、第 6 章基于 NRZI 信息的练习部分）。USB 包被分为令牌、数据和握手，如图 9-27 所示。每个包以一个同步序列开始，以一个包结束（EOP）标记结束。USB 1.x 的包以一个 8 位同步序列（如 00000001）开始，以一个 EOP 标记结束，EOP 将保持 D+ 和 D- 信号 2 位时长为 0。另一方面，USB 2.0 标准使用 32 位（4B）同步序列和 8 位 EOP 标记。

9.8.2 事务

每个 USB 事务包含一个或多个包。例如，一个发送打印数据给打印机的事务要求，按顺序，为令牌包、数据包和握手包。

如图 9.27 所示，令牌包和数据包都包含一个类型域、一个数据域和一个放置错误检测

代码的域。握手包只有一个类型域。一个令牌包用于确定 4 种事务中的一种，4 种事务分别为帧起始（SOF）、建立（SETUP）、输入（IN）和输出（OUTPUT），如表 9-6 所示。一个数据包包含一个有效负荷并且其类型为数据 0 或数据 1。USB 2.0 有其他的数据类型，例如数据 2，其用于告知将被多个事务传输的大有效负荷的大小。

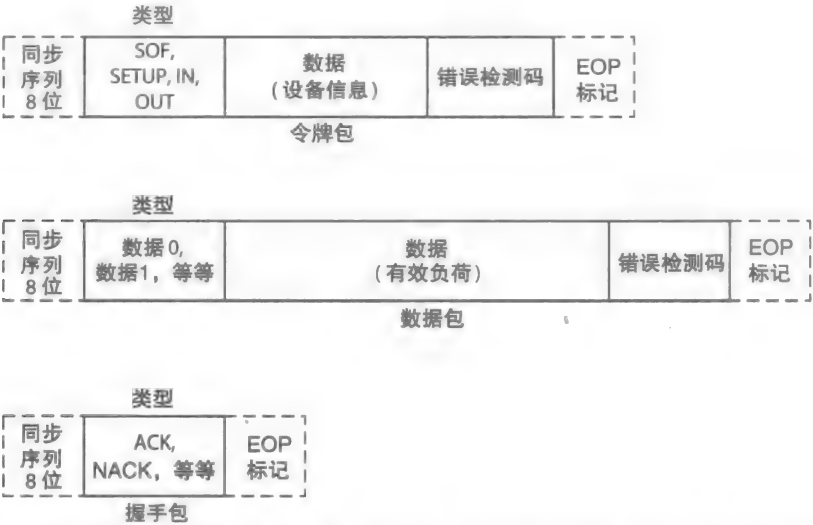


图 9-27 三种类型的 USB 通信包。每个包以一个同步序列开始，以一个 EOP 标记结束。

如果一个事务包含了一个从 / 到设备的全部的有效数据负荷，其数据包类型将为数据 0。另一方面，如果使用两个或多个事务来传输一个大负荷，连续的数据包将被交替标记为数据 0 或数据 1。每个源模块和目的模块包含一个触发器，其初始化为 0。

当源模块发送或目的模块接收一个数据 0 类型或数据 1 类型的包时，其触发器将触发，如果为 0（表明数据 0），其将变为 1（期待下一个接收包的类型为数据 1），若为 1（表明数据 1），其将变为 0（期待下一个接收包的类型为数据 0）。这保证了在目的模块的数据包只会被接收一次，以防止发生错误，例如，当一个目的模块正确接收到一个数据包并且触发了其触发器，但是源模块认为并不是这样，保持其触发器不变并对该数据包进行重传。在这种情况下，因为两个触发器不再相同，目的模块将拒绝接收重传的数据包。握手包有多种用途。例如，一个确认包（ACK）是用于确认接收到数据包，非确认包（NACK）是用于控制数据流，例如，当一个设备接收到一个要求其发送数据的包但其无数据发送时。

表 9-6 USB 包标识符示例

| 包 | 类 型 | 描 述 |
|----|-------|-----------------|
| 令牌 | SOF | 帧开始包用于同步数据 |
| | SETUP | 建立事务用于设备配置 |
| | IN | 输入包用于从设备输入数据 |
| | OUT | 输出包用于输出数据到设备 |
| 数据 | 数据 0 | 当发送器的触发位为 0 时使用 |
| | 数据 1 | 当发送器的触发位为 1 时使用 |
| 握手 | ACK | 表明数据包正确接收 |
| | NACK | 表明设备当前无法发送或接收数据 |

9.8.3 传输

数据传输的 4 种类型，分别为**中断**、**准同步**、**批量**和**控制**，如表 9-4 所示，其用于给所有类型的 USB 设备提供服务。正如早先讨论的那样，USB 设备并不采用中断驱动传输机制，相反，其使用中断包来轮询如键盘、鼠标等在旧式计算机中采用中断驱动传输机制的 USB 设备。在这种情况下，一个设备，如键盘，在轮询到该设备但其无需发送任何扫描码时，设备将发送一个 NACK 握手包。

实时 USB 设备，例如数字电话，必须保证其可以发送或接收数据，当扬声器正在使用时，其必须不被中断从而频繁接收数据。这些设备使用同步传输，相应的事务包含令牌包和数据包，但没有握手包，这些事务的数据不会进行重传。数据块传输应用于诸如打印机、传真机、扫描仪和绘图机等需要精确数据传输并可在传输数据时接受中断的设备。控制传输在设备或集线器配置时使用。

多个设备可使用一个主控制器接口来发送或接收数据。然而，一个设备并不允许连续不断地进行通信，因为这会导致其他设备产生饥饿现象，影响其他设备的正常运转。例如，考虑图 9-18 中的系统，假定用户在使用扬声器听歌时还希望打印一个文件，打印机和扬声器必须共享主控制器来分别接收打印数据和音乐数据。如果主控制器接口允许一段时间连续不断地传输打印数据，音乐数据的传输将被中断，导致音频质量很差。相似的，如果主控制器允许连续不断地传输音乐数据，打印工作可能无法完成。在另一个场景中，用户可能编辑一份文档，同时打印机打印数据并且扬声器播放音乐。在这种情况下，键盘和鼠标也必须可以及时和系统进行通信。

9.8.4 描述符

每个 USB 主控制器接口实现一个或多个**接入点**，每个接入点也称作**终端**。每个终端包含一系列可被主控制器访问的 I/O 端口，每个终端包含一个描述符，该描述符被称为**终端描述符**，用于确定传输类型——例如，控制、中断、准同步或数据块——也用于确定其他需求，例如在每个数据包中允许传输的最大数据量。控制、中断和数据块传输的最大数据包为 64B，另一方面，同步传输使用更大的数据包进行传输（例如，在 USB 2.0 中为 1024B）。SUB 集线器控制器实现**状态更改端点**，其每 255ms 轮询一次，来检测可能发生的端口事件，例如从 USB 端口连接或断开连接等。

一个设备可以包含多种类型的终端从而可实现不同种类的接口。例如，USB CD-ROM 驱动器可实现 3 种不同的接口：供读写文件的大数据存储接口，处理音乐的音频接口，处理视频镜像的视频接口。在每个 USB 设备中的接口包含一个**接口描述符**，其包含多个终端描述符。接口描述符是**配置描述符**的一部分，配置描述符包含多种配置，包括例如当系统电池供电时的低功率配置。所有这些描述符都是在一个层次结构中组织起来的，并且包含在一个**设备描述符**中，设备描述符也包含设备信息，例如制造商名称和序列号。每当集线器轮询检测到有新的即插即用设备时，USB 驱动将访问设备描述符。描述符包含所有必需的设备信息，包含一系列在设备内的 I/O 端口相对地址和设备传输数据的传输类型。驱动将给该设备分配一个唯一的地址并配置该终端。

[417]

9.8.5 帧

USB 主控制器接口设计用来服务当前与系统连接并活跃的所有设备，其每隔一定时间

发送和接收的包被称为帧。例如，USB 1.x 使用 1ms（毫秒）帧，USB 2.0 使用 125 μ s（微秒）帧，被称为微帧，对于 USB 1.x 的全速设备，每帧为 12 000bit（12Mbit/s * 1 ms \approx 12 000bit）长；对于 USB 2.0，一个微帧为 60 000bit（480 Mbit/s * 125 μ s \approx 60 000bit）长。

假定图 9-18 中的系统使用单个 USB 主控制器接口，图 9-28 说明了一个 USB 1.x 主控制器可能的帧内容。在图中，每个帧包含对多个事务的包，并以 SOF 事务开始。例如，考虑以下场景，用户同时使用数字电话通信、编辑文档、听音乐、打印文档、发送或接收传真和写 flash 存储器。在这种情况下，帧内容将包含处理下列事务的数据包：

两个与数字电话相关的同步音频 IN 和 OUT 事务。

一个与扬声器相关的准同步立体声音频事务。

一个与键盘相关的中断 IN 事务。

一个与鼠标相关的中断 IN 事务。

零个或多个与集线器相关的控制事务。

如果可能，一个与打印机、传真机和 flash 存储器相关的数据块 OUT 事务。

与中断、准同步、控制传输相关的数据包具有较高的优先级，其首先加入每个帧中，只有剩余足够的带宽时，才会将数据块传输相关的数据包加入每个帧中。

[418]

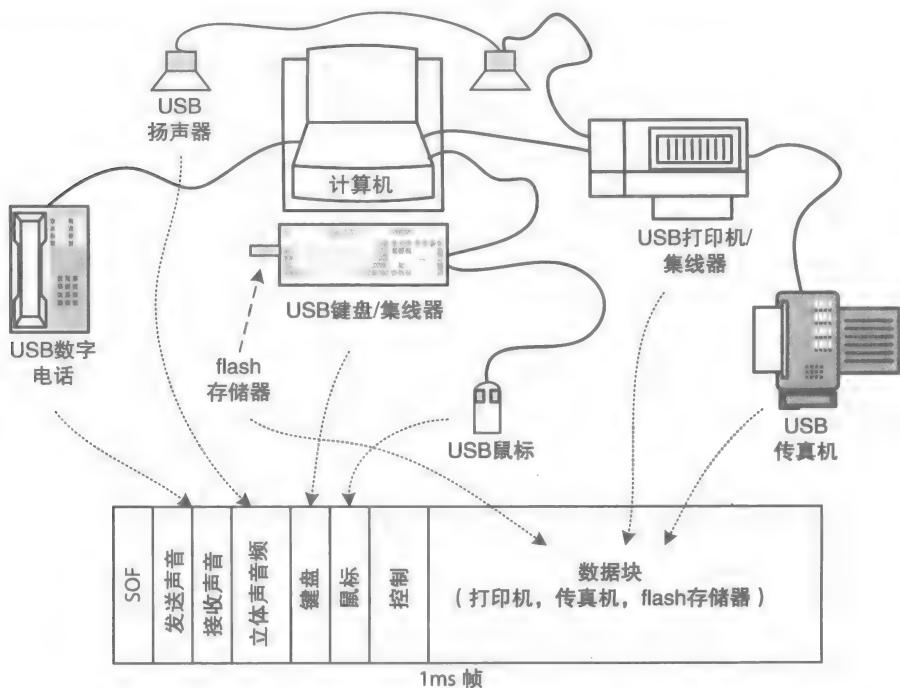


图 9-28 在一个帧中的 USB 传输，对于 USB 1.x 帧持续时间为 1ms[4]

总体而言，除了那些需要中断传输或同步传输的设备，其他设备并不是在每个帧中都有数据包。一般情况下，帧带宽的 90% 将被保留以供中断传输或同步传输，10% 用于控制传输，0% 的帧带宽用于数据块传输。任何时间发生中断传输或同步传输时，数据块传输将不被允许，这将导致使用数据块传输的设备运转出现短时间的停顿。此外，当出现一个新的附加设备时，若其传输类型需要比当前存在设备可能的带宽还大时，该设备将不会被配置。

USB 2.0 主控制器使用分离事务来与通过高速集线器与控制器接口相连接的低速设备或

全速设备进行通信。在这种情况下，一个低速 OUT 事务或全速 OUT 事务将被分离为多个 **start-split (SS)** 微帧事务。SS 事务中的数据将被存储在高速集线器中的缓冲器中，再发送给低速设备或全速设备。一旦一个目标低速设备或全速设备接收到一个 IN 请求，其使用更低的数据包传输速率 (1.5Mbit/s 或 12Mbit/s) 来传输数据到高速集线器，高速集线器接收到数据并放置其到输入缓冲器中。在缓冲器中的数据之后通过被称为 **complete split (CS)** 的事务传输给 USB 2.0 主控制器。

419

9.8.6 事务组织结构

每个 USB 设备都有一个驱动例程，该例程被称为**客户端驱动**，其通过两个主机软件 (**USB 驱动**和 **USB 主控制器驱动**) 来与设备通信。客户端驱动明确与设备通信的内容，并向 USB 驱动发起请求。每个客户端将申请供与设备通信相关数据存储的存储器空间。例如，一个 USB 键盘 (客户端) 驱动发起一个中断传输请求并提供一个供键盘数据 (扫描码) 存储的存储器地址。USB 驱动将每个客户端请求转换为一个或多个 USB 事务描述符 (TD)。每个 TD 是一个数据结构，其包含与传输、与下个 TD 相连和保存客户端 IN 或 OUT 的数据的存储器空间等相关的必要信息。

活跃的所有客户端的 TD 将被分组并在存储器中组织成 TD 的一系列链表。在 USB 2.0 中的控制 TD 和数据块 TD 将被组织在一起，而中断 TD 和同步 TD 将被组织在一起，如图 9-29 所示，中断 TD 和同步 TD 以链表数组的方式组织起来并周期性地进行处理，另一方面控制 TD，图中为 qTD，以链表队列的方式组织起来并不定时间来处理。每个 iTD 链表在一个 125 μ s 微帧内执行。当数组中没有未执行的 iTD 时，qTD 将被执行。

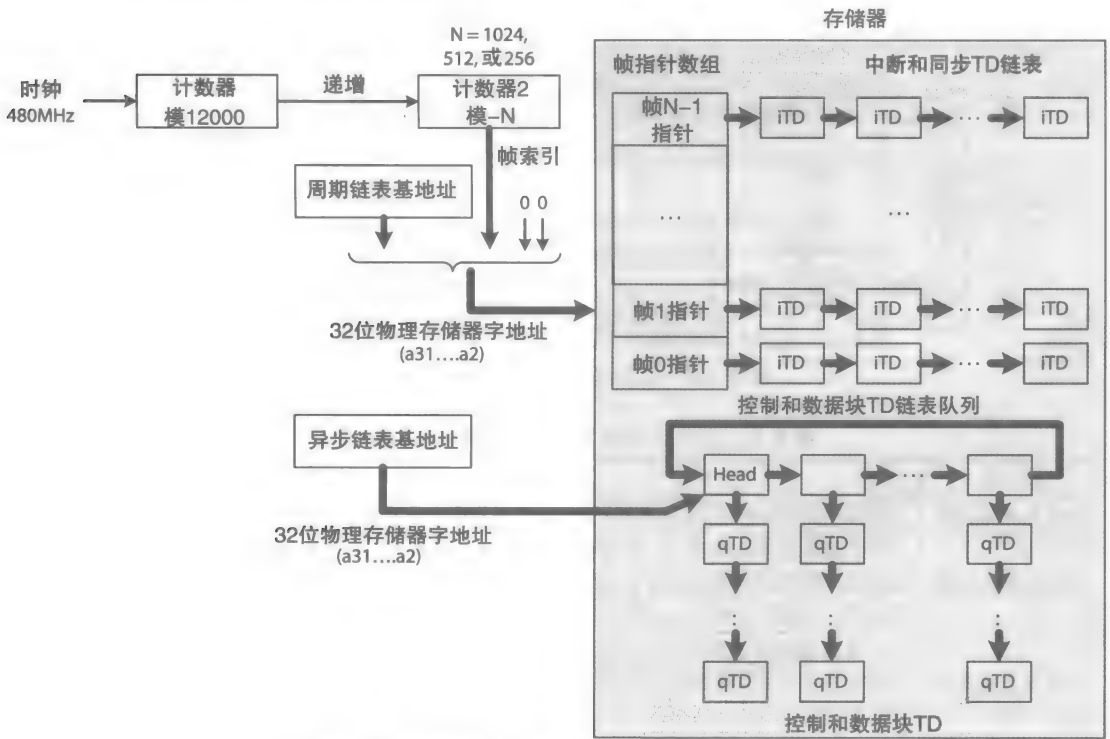


图 9-29 存储器中 USB 2.0 事务组织结构 [4]

USB 驱动通过 USB 主控制器驱动存储数组的起始地址和队列的头到 USB 主控制器接口中。USB 主控制器使用两个计数器来执行 iTD，如图所示。计数器 1 以 480MHz 时钟模 60 000 方式计数，计数器 2 保持当前帧数并用作数组的索引。计数器 2 每 125 μ s 增加一次。数组的大小是可编程的，可存储 1024、512 或 256 个元素。数组的基地址存储在一个称为周期链表基地址的寄存器中，其与计数器 2 相连来确定下一个事务在存储器中的位置。异步链表基地址指向存储器中链表队列的头，如图所示。

每当计数器 2 增加时，将执行一次 SOF 事务，其通知所有使用同步传输的设备对其活动进行同步。如果一个 TD (iT D 或 qTD) 描述了一个 OUT 事务，主控制器将从主存储器中获取其令牌和数据包并将其传输到目标终端。如果终端采用中断、控制或数据块传输，控制器将从终端接收到一个握手包。正如早先说明的那样，同步传输不需要握手包。

如果一个 TD 是一个 IN 事务并标明其是中断、控制或数据块传输，主控制器将从存储器中获取 TD 的令牌包和握手包，并传输令牌包到目标终端。当控制器接收到来自终端的数据包时，若接收的数据没有错误，控制器将向终端传输一个握手包。

9.8.7 事务执行

和其他 DCI (如图 9-17) 相似，USB 主控制器接口包含了一系列 CPU 和主控制器 (嵌入式系统) 访问相关的 I/O 端口，如图 9-30 中关于 USB 2.0 的主控制器接口。没有设备数据是通过 I/O 端口直接与 CPU 进行通信的；端口只用于配置和建立主控制器。表 9-7 列出了一系列 USB 2.0 主 I/O 端口，其被称为 USB 操作寄存器。这些寄存器用于设置帧数组的大小为 1024、512 或 256，中断 CPU 的频率等，并根据微帧的数量指定中断频率。微帧数量可参考给定的中断阈值来确定为 1、2、4、8、16、32 或 64 微帧。

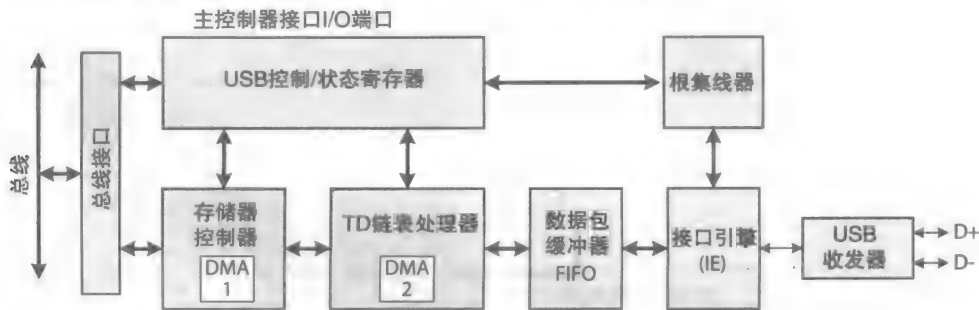


图 9-30 单端口 USB 2.0 EHCI 框图 [6]

表 9-7 USB 2.0 操作寄存器 (如 I/O 端口) 列表

| 寄存器 | 描述 |
|-----------|--|
| USB 基址寄存器 | 维持一个基址寄存器以供存储器对主控制器接口内的 I/O 端口进行映射 |
| USB 命令寄存器 | 用于配置主控制器接口。配置如中断间隔 (如每个微帧、每两个微帧、每 4 个微帧等)、开始或关闭周期性的或异步的 TD 链表处理、初始化帧数组大小 (1024、512 或 256)、主控制器接口复位、运行位或停止位等 |
| USB 状态寄存器 | 包含周期的和异步的使能/禁用位、主控制器接口停止/运行状态位、USB 错误中断位、USB 中断位 (如果中断完成 (IOC) 位在 TD 中被设置, 该位将变为 1)、异步中断状态位 (当主控制器接口增加队列指针时, 其将被设置) 等表示 USB 信息的状态位 |
| USB 中断使能 | 开启多个中断, 常用的有开启异步链表中断, 使得 USB 驱动可知道当前队列状态; 使能错误中断等 |

(续)

| 寄存器 | 描 述 |
|---------|--|
| USB 帧索引 | 保存一个与帧数组相关的地址 (索引), 每 125 μ s 增加一次。其表明图 9-29 中帧数组的大小 |
| 周期链表基地址 | 帧数组的基地址。其与帧索引一起确定帧数组在主存储器中的地址 |
| 异步链表地址 | 指向队列头的指针 (图 9-29) |
| 端口状态和控制 | 用于配置 USB 端口。如开启 wake-on-connect 和 wake-on-disconnect 位, 用于检测低速、全速、高速设备连接的线路状态 (D+ 和 D-); 当前端口连接状态 (如 1 表示设备连接, 0 表示设备断开) 等 |

例如, 如果中断阈值设置为每 2 个微帧一次, 主控制器将在 2 个微帧内先执行所有的 TD (iTD 和 qTD), 其中先执行 iTD。如果 interrupt-on-complete (IOC) 位在一个或多个 TD 中被设置, 主控制器将在第二个微帧末向 CPU 发出请求来中断 CPU。作为主控制器接口配置的一部分, CPU 可能选择开启或关闭 TD 的执行。

当处理器完全配置好主控制器接口时, 其将设置 run 位——与处理器初始化 DMA 传输相似——该位在 USB 命令寄存器中, 设置该位将开始主控制器并开始 TD 的执行。先进先出存储器缓冲器是用于在数据传输前存储输出的数据包, 也用于在数据发送给主存储器前存储从设备接收到的数据。在这种情况下, 存储器控制器将负责生成存储器地址以获取 TD、从 / 到主存储器中读或写终端数据, 向主存储器写状态数据。例如, 在图 9-30 中, DMA-1 用于从主存储器中获取 TD, 存储 TD 到链表处理器内部的 RAM 中, 也用于获取其输出到终端的数据, 通过 DMA-2 将数据存储到 FIFO 缓冲器中。DMA-2 用于从 FIFO 缓冲器中接收终端输入的数据, 并通过 DMA-1 传输到主存储器中。

DMA 控制器用于当前的处理任务, 其提升了在主控制器接口的数据输入输出速度。包含一系列配置寄存器的根集线器负责端口的管理, 例如端口复位、唤醒等工作, 也检查端口连接或连接断开。接口引擎 (IE) 负责 NRZI 数据编码和译码、令牌构建等工作。

参考文献

422

1. William Stallings, *Computer Organization and Architecture*, Prentice Hall, 8th ed., 2010.
2. Mobile Intel Pentium Processor with 533MHz Front Side Bus, <http://www.intel.com/Assets/PDF/datasheet/253028.pdf>.
3. Microcontroller, <http://www.atmel.com/products/>, <http://www.cypress.com/products/>.
4. Don Anderson and Dave Dzatko, *Universal Serial Bus System Architecture*, 2nd ed., Addison Wesley, 2002.
5. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: Systems programming Guide, Part 1, 2009.
6. USB 2.0 host controller core (inSilicon), <http://www.synopsys.com/>.

练习

- 9.1 考虑一个 FSB, 其连接了 4 个 1GHz 处理器和一个共享的存储器单元, 生成一个 UMA 架构。假定指令中 10% 是写数据到存储器的 “ST” 指令, 完成以下:
- a. 假设 CPI 为 1, 每次写 4B, 确定 4 个处理器写数据要求的存储器写带宽。
 - b. 考虑存储器也需要提供指令和数据给处理器, 确定达到 (a) 中两倍带宽所需的 FSB 的时钟频率, 假设 FSB 运行与 SDRAM 总线相似, 突发大小为 1, 32 位数据总线。
 - c. 假设存储器控制器操作一个 32 位 400MHz 的 SDRAM 存储器单元, 确定达到峰值性能的存储

器组织结构, 其中每个 FSB 时钟周期传输 4B。

- 9.2 研究 Intel 的快速通路链接并以此写一个短论文。
- 9.3 研究 AMD 的超传输通道并以此写一个短论文。
- 9.4 绘制一个含有 6 个节点 9 个串行链路的 NUMA 结构图, 其中每个处理器与其他处理器之间不超过 2 条串行链路。
- 423
- 9.5 考虑一个采用串行内部互连的 4 处理器 NUMA 结构与一个基于总线的 4 处理器 UMA 系统。回答以下问题:
- 比较串行链路与总线互连的优劣。
 - 当处理器数量增加时, 比较 NUMA 与 UMA 架构的优劣。
- 9.6 考虑图 9-6 中的存储器控制器。假设 SRAM 读写访问时间为 4ns, CPU 总线时钟频率为 0.5ns, 其需要一个时钟来检测到 $ack = 1$, 并且需要 1 个时钟周期来完成存储器周期, 从总线上载入数据到内部寄存器并结束读周期或移除总线上的数据并结束写周期。假设存储器读和写的访问时间相同, 确定计数器的大小。
- 9.7 假设一个新的 Samsung 磁盘驱动器的 RPM 为例 9-1 中的两倍。确定在存储器和磁盘驱动器间传输 512B 数据的时间。
- 9.8 图 9-10 中绘制了存储器映射的 I/O 端口的电路, 绘制一个端口映射 I/O 的电路。
- 9.9 考虑 9-12 的时序图, 完成以下:
- 简单解释为何当 $sel = 1$ 且 $_{wr} = 0$ 时, 输出端口将载入总线上的数据。
 - 假设输出端口采用触发器来设计。接入和运行端口有何区别? 绘制输出端口并解释输出端口是何时、如何从总线上载入数据。注意: 端口触发器并不是以连续变化的时钟信号来运转的。
- 9.10 假设图 9-17 中的端口 0 和端口 1 (4-B 端口) 采用存储器映射 I/O, 端口地址为 0x60 和 0x64。为两个端口设计地址译码电路。为了简单起见, 假设地址总线宽度为 8 位。
- 9.11 解释为何一个处理包含 DMA 传输信息的数据结构的现代 DMA 控制器比图 9-20 中的简单 DMA 控制器要好, 并说明现代控制器是如何影响系统性能的。
- 9.12 解释为何当 CPU 接收到一个比当前执行的中断处理程序 (IVc) 小于或等于的 IV (IVr) 时, CPU 将忽略 IVr 直到 $IVr > IVc$ 。
- 9.13 假设一个 NAND 门延迟是 0.1ns, 一个三态缓冲器延迟为 0.2ns, 估计一个从 16 节点的菊花链式中断结构的 IV 放置到总线前的最坏时间延迟。
- 9.14 Sparc CPU 实现了 8 窗口的寄存器窗口, 其指的是 CPU 数据通路包含所有用户可访问的寄存器的 8 个副本。例如, 图 9-26 中的 ACC、X 和 SR 是用户可访问的寄存器。对于一个寄存器窗口, 当有子程序调用或中断发生时, 当前执行程序的状态将被保存在 CPU 中而不是存储器中。完成以下:
- 假设图 9-26 中的 CPU 含有 4 个寄存器窗口, 解释寄存器窗口是如何给等待设备提升服务的。
 - 假设图 9-26 中使用 ACC、X 和 SR 寄存器的 4 个副本从而产生大小为 4 的寄存器窗口。并假设对于窗口 0 这些寄存器命名为 ACC0、X0 和 SR0, 对于窗口 1 这些寄存器命名为 ACC1、X1 和 SR1 等。当发生中断时, 当前执行的下一条指令需要使用一个寄存器, 例如, 窗口 1 中的某个寄存器。例如, 当 IH 中的指令 “MVSXR2ACC” 执行 $ACC1 \leftarrow SRX$ 时, 被中断的程序的状态将依旧保存在窗口 0 中寄存器 ACC0、X0 和 SR0 中。RTI 将切回至窗口 0, 从而被中断的程序可使用窗口 0 中的寄存器来恢复执行。设计一个 4 窗口寄存器窗口电路并简单描述其是如何切换窗口的。

- c. 讨论如何扩展 (b) 中的寄存器窗口使之可通过 ACC 传输单个参数给子程序从而支持子程序调用。并且说明当有多级子程序调用时如果运行。
 - d. 寄存器窗口提供了特定的优势，假设 CPU 有 8 个寄存器窗口。程序员可如何利用寄存器窗口的优势？
 - e. 研究 Sparc 处理器如何使用重复的寄存器窗口来传输参数，并以此写一篇论文。
- 9.15 简单说明为何 USB 主控制器需将数据包封装成定期从设备传输或传输到设备的帧。
- 9.16 简单说明为何图 9-29 中的 iTD 要在 qTD 前处理。
- 9.17 简单说明为何图 9-30 中的 DMA-1 和 DMA-2 是必需的。
- 9.18 简单说明在 USB 主控制器接口内部中将异步链表提前中断的目的。(提示：现代 DMA 控制器如何工作。)

计算机安全

- 9.19 计算机安全 (安全中断)：见练习 11.36，当中断返回时关于检测寄存器欺骗、拼接或重放攻击 (也可参见 11.11.9 节)。

存储系统

10.1 简介

统一内存访问（UMA）和非统一内存访问（NUMA）系统的性能都由延迟决定。对包括存储器延迟在内的任何延迟上的改进都会增大系统的带宽。CPU（处理中心）处于空闲状态的时间越久，执行程序所需的时钟周期数就会增加，从而降低了指令的吞吐率。此外，系统用来存储程序 and 数据的非易失性存储和主存的容量都必须足够大，以便使它能并发地运行系统和应用程序。这意味着存储器必须具有成本效益。然而，当前还没有任何可用的技术能被用来制造低延迟、大容量以及低价位的存储器。

现今普遍使用的存储器技术是静态随机存取存储器（SRAM）、同步动态随机存取存储器（SDRAM）、磁性存储器和闪存。SRAM 技术最快，但也是最昂贵的，因此被用作处理器中的 cache。SDRAM 技术比较便宜但也相对较慢，需要的访问时间大约为 100 个 CPU 周期。磁性存储器和闪存是非易失性的并且也是最便宜的，但是它们速度最慢，而且需要近乎毫秒级的访问时间，大约比 CPU 慢 1 000 000 倍。

然而，因为程序包含循环，而且数据通常都是从存储器中顺序访问的。因此，通过结合各种技术，如图 10-1 所示分层组织，最慢存储器在底部，最快存储器在顶部，才能创建一个减小平均延迟、降低花费并拥有大存储容量的存储系统。注意，在第 8 章作为 CPU 数据通路的一部分被引入的指令存储器（IM）和数据存储器（DM）被重新标记为指令缓存（Ic）和数据 cache（Dc）。

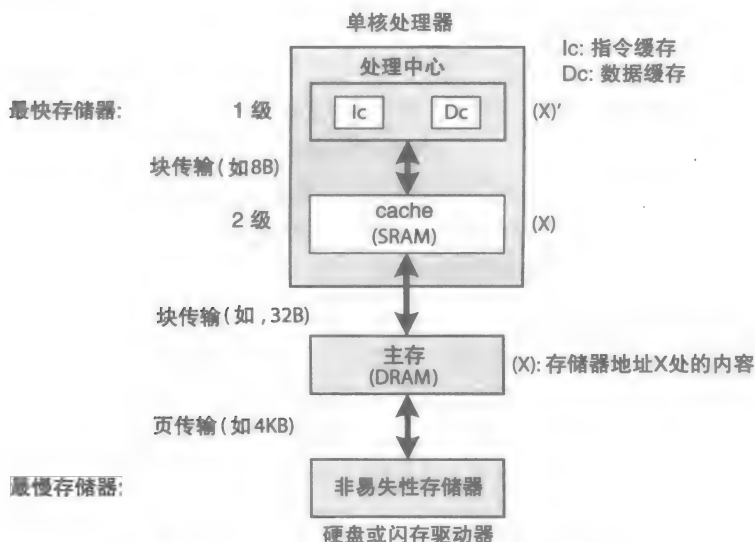


图 10-1 一个 4 级存储器分层的存储系统

其他存储系统的设计目标可能包括低能耗、高可靠度和小型号。低能耗和小型号的需求

对于手持设备尤其重要。

在数据被 CPU 访问之前，它从存储器层级的最底层被复制到上一层，直到最高层为止。此外，多个数据项（作为一个块或一页）在层级之间相互复制。而且，只要数据在主存储器中，CPU 将会等待接收数据。正如我们在 10.4 节所见，如果 CPU 请求的指令或数据不在主存储器中，程序将停止运行。

当程序运行时，修改过的数据从存储器层级的较高层被复制到较低层，从而创建空间或在必要时通知低层存储器这些改变。这就会造成同一个数据在不同两个存储器中的两个副本可能不相同的情形。例如，考虑在图 10-1 中，存储器地址 X 处的一个数据项从主存被复制到 2 级（L2）cache。在被 CPU 访问之前，数据接着从 L2 cache 被复制到 1 级（L1）数据 cache。现在假设 CPU 在操作完该数据项后写了一个新值到 L1 cache 中，改变了副本，在 cache 中用（X）' 表示。L1 cache 中该数据项的副本现在则与仍然储存在 L2 cache 和主存中的副本不同了。

此时此刻，假设操作系统（OS）命令直接内存访问（DMA）控制器从主存中传输包括地址 X 处的数据在内的数据到硬盘中。显然，应该被复制到硬盘中的数据项必然是处于 L1 cache 中那个改变过的数据项。同样在另一个场景中，OS 可能会命令 DMA 控制器用硬盘上的副本更新主存，包括地址 X 处的内容。这次，主存中的副本会是最新、有效的，而两级 cache 中的副本则是旧的、无效的。因此，为了防止陈旧数据被保存在硬盘上或被 CPU 访问，必须实现某些在存储器层级之间的数据一致性协议。

此外，由于与非易失性存储器（硬盘或闪存驱动器）的速度对比，易失性存储器的访问时间非常少（最多相当于 100 个 CPU 周期），在除了最低两级之外的存储器层级之间复制并保持一致性的任务要在硬件上进行。在被称为虚拟内存的非易失性存储器（例如硬盘）和被称为物理内存的主存之间复制并保持数据一致性，需要在软件上执行一些功能并在硬件上进行一些操作。虚拟内存管理系统决定了物理内存的何处用来存储硬盘上一个或多个程序的代码及数据。

本章展示了存储器层级的体系结构和组织并提供了可选的缓存组织及一致性协议。根据需求，缓存可以被设计为最大限度地减少硬件和访问时间或流量。还讨论了能耗以及 cache 组织和系统性能之间的关系。

本章还展示并用图说明了虚拟到物理地址的映射方案，讨论了在处理器芯片的何处必须实现这种映射方案并给出了可选的解决方案。

存储器层次结构

在图 10-1 中，低成本的非易失性存储器被用来构建潜在的无限制存储器容量。DRAM 技术、典型的 SDRAM 被用来构建大型主（物理）存，SRAM 技术被用来构建处理器中小而快的缓冲存储器。

在程序执行过程中，程序代码和数据从非易失性存储被复制到主存，然后从主存到 L2 cache，接着到两个 L1 cache。较高层存储器只会保存低一级存储器内容的一个小片段。此外，由于 DMA 传输（第 9 章）和突发存储器传输（第 7 章）更有效率，页面（例如，4KB）在非易失性存储和主存中进行传输。也被称为 cache 块或 cache 行的块（例如 32B 或 64B）在主存和 L2 cache 之间传输，甚至更小的 cache 行在 L2 cache 和每个 L1 cache 之间传输。指令 cache 行从 L2 cache 传输到 Ic，数据 cache 行在 L2 和 Dc 之间传输。

CPU 中访问内存的请求, 若其地址指向一条指令则总是首先到达 Ic, 若请求数据则先到 Dc (可见于第 8 章的图 8-5)。如果目标内存块的副本在 Ic 或 Dc 中, 该次访问称为 **cache 命中**。反之, 该次访问叫作 **cache 缺失**, 然后请求被 Ic 或 Dc 转发到统一的 L2 cache, 它同时包含指令和数据。同样, 如果块的副本在 L2 cache 中, 则该次访问称为 **cache 命中**, 否则称为 **cache 缺失**。如果是 **cache 命中**, 若块的副本为指令, 则 L2 cache 将其传输到 Ic 否则传输到 Dc。另一方面, 若该次访问是 **cache 缺失**, L2 cache 将请求发送到主存。

429 在现代计算机系统中, 程序通常都不是整个被载入主存中的。反之, 程序和数据页面需要时才从非易失性存储里被复制到主存中。修改过的数据页面被写回到非易失性存储以便释放主存空间。然而, 指令页面若在一段时间内不被引用并且需要内存空间则直接被丢弃。这个过程称为**页面调度**, 这牵涉到了 OS, 而且需要硬件来实现**虚拟内存组织**, 10.4 节将会进行讨论。

特别地, 在 L2 cache 缺失的情况下, 如果主存中的一个页面包含目标指令或数据块, 当前块的副本会被传输到 L2 cache 中。否则, 若主存中没有该页面, 该请求将会导致页面错误中断 (第 9 章), 程序的执行将会暂停并将控制权交回给 OS。一旦页面被载入主存中, 程序恢复执行。

正如第 8 章所探讨的, 在 **cache 缺失**得到解决的整个过程中, 处理中心 (CPU) 处于空闲状态且不执行指令。实现了多线程技术的内核可以切换到另一个不同线程中执行指令。如图 10-2 所示, 现代处理器通常设计了 3 级 **cache** 存储器。每个处理中心与各自的 L2 cache 通信, 而所有的 L2 cache 与共享的 3 级 **cache** 通信, 从而在处理器芯片中构建了一个 **UMA** 系统。共享 **cache** 的优势在于共享的 **cache** 行能被两个或更多线程使用, 但也有其缺点, 即某个线程删除了另一个线程正在使用的块。

例如, 考虑 8.4.4 节 (第 8 章) 的程序示例中线程 0 和线程 1。回忆操作不同数组元素的每个线程, 线程 0 计算得到的和为 $sum[0]$, 线程 1 得到 $sum[1]$ 。线程 1 接着输出 $sum[0] + sum[1]$ 。通过共享的 L3 **cache**, 线程 1 能够访问到 L3 中的 $sum[1]$, 节省了一次主存访问。总的来说, 通过 L3 共享 **cache**, 不仅两个交互线程的执行会更快, 而且执行过程会产生更少的主存流量。Intel Xeon-E7-4870 (Nehalem 结构) 处理器含有 10 个内核, 每一个都连接到一个 256KB 的 L2 **cache**, 然后连接到 30MB 的共享 L3 **cache**。该处理器能访问最大 32GB 的主存空间。

430

存储器延迟

存储器层级减小了平均延迟, 因为程序包含循环并且数据访问通常都是顺序的。考虑图 10-1 所示的存储器层级, 假设主存中的一个块含有一个小型 **for** 循环指令。当 **for** 循环第一次执行时, 获取第一条指令的请求会导致一次 **cache 缺失**。这个块会从主存复制到 L2 **cache** 然后到 Ic。因此, 从主存传输一个块的副本到 Ic 的延迟很长。

然而, 一旦该块被载入 Ic (一个较小的 SRAM), 块中随后的指令会更快 (例如, 在 1

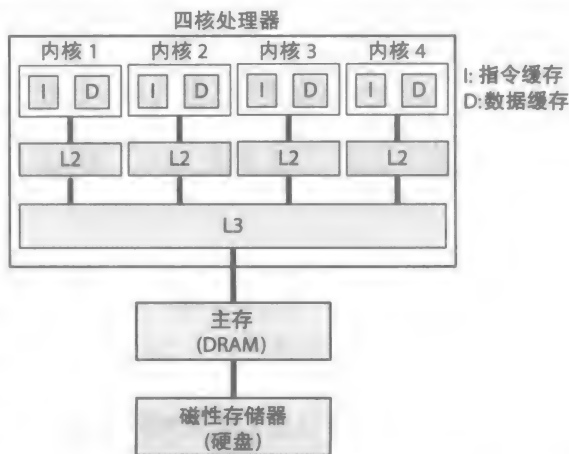


图 10-2 一个 5 级存储器层级的例子

个 CPU 时钟周期内) 从 Ic 中被获取。因此, 这减少了执行 for 循环的平均存储器延迟。这与当顺序从主存中访问数据时相同。访问导致缺失的块中的第一个数据项会花费较长时间, 但从 Dc (也是一个较小的 SRAM) 中访问块中随后的数据将会更快。

总之, 程序中每条指令再次执行的速度表明了程序中存在的**时间局部性**的多少。因为小型 for 循环中的指令执行地很频繁, 一旦一条指令执行了, 很快会再次执行 (由于是小型循环), for 循环中存在很高的时间局部性。

与此类似, 程序在执行中数据结构元素被访问的顺序表明了程序中**空间局部性**的多少。例如, 考虑一个处理数组的 for 循环。在 for 循环执行期间, 如果数组元素是从连续的内存地址中访问的, 那么从主存中访问的这些元素之间具有很高的空间局部性。

注意, 时间局部性也能应用到程序数据——例如, 如果在循环中访问相同的数据元素集。同样, 空间局部性可以应用到程序指令。

cache 命中根据导致命中的一次访问的概率来决定。例如, 当执行一个程序时, 如果 95% 的时间里指令都在 Ic 中找到了, 那么指令**命中系数** (也称为**命中率**) 是 0.95, 使得指令**缺失系数** (也称为**缺失率**) 是 0.05 ($1.0 - 0.95$)。数据在 Dc 中被找到的频繁程度决定了程序数据的命中率和缺失率。L2 和 L3 cache 是统一的 (都包含指令和数据)。因此, 它们为整个程序提供命中率和缺失率。命中率的高低决定于程序中存在的**时间局部性**和**空间局部性**的多少。

一个好的程序员在写程序时应该多留心时间和空间局部性, 同时在程序执行过程中的数据的空间局部性存在可能的改进时, 编译器会重排代码。

例 10-1 考虑图 10-1 所示的存储器层级。给定下列信息, 估算它的平均存储器延迟。同时假定了主存带宽的峰值。

Ic 和 Dc: 延迟 = 1ns, 命中率 = 0.95 (假定两个 cache 相同)

L2 cache: cache 行 = 32B, 延迟 = 3ns, 命中率 = 0.9

主存: 400MHz 的 SDRAM, 32 位 (4B) 的数据总线

431

解: 为了达到性能峰值, 忽略激活行和发布列地址的时间。这将会在当 SDRAM 存储器访问多个块的操作重叠时发生。例如, 考虑如图 7-19 (第 7 章) 所示的 SDRAM 定时图, 其中大小为 4 的两次单独的突发访问 (每次形成一个块) 是从存储器中顺序读取的。在图中, 两个块的第一个数据项分别标记为 x 和 y 。注意到存储器操作重叠了。在第一个数据项 x 被访问之后 (如出现在数据总线上), 每个数据总线时钟周期访问一个新的数据项。访问数据项 y 、 $y + 1$ 、 $y + 2$ 、 $y + 3$ 的总时间相应的只需要 4 个数据总线时钟周期。如果存储器在每个时钟周期内足够提供一个数据项, 存储器就运转于带宽峰值。正如接下来计算的, SDRAM 能提供 1.6GB 的峰值传输量。因此, 从主存传输 32B 的块到 L2 cache 中将会花费 20ns。

$$\begin{aligned}\text{主存峰值带宽} &= 400\text{M cycles/sec} * 4\text{ B/cycle} \\ &= 1.6\text{GB/s} \\ \text{主存延迟} &= 32\text{B}/1.6\text{ GB/s} \\ &= 20\text{ns (只是传输时间)}\end{aligned}$$

目标指令块在 Ic cache 中和目标数据块在 Dc cache 中的概率是 95%。在 5% ($100 - 95$) 的时间里目标块不在 L1 cache (Ic 或 Dc) 中, 同时该块在 L2 cache 中的概率是 90%。最后, 该块不在 L2 中的概率是 10% ($100 - 90$), 此时该块在主存中。正如之前所述, 如果该块不在主存中, 程序将会停止执行, OS 会分配 CPU 去执行另一个程序。因此, 下面所计算的预计

平均存储器延迟不（也不应该）包括页面延迟。然而，页面延迟包含在程序整个执行时间里。

$$\begin{aligned}\text{平均延迟} &= (0.95)(1\text{ns}) + (1 - 0.95)(0.90)(3\text{ns}) + (1 - 0.95)(1 - 0.90)(20\text{ns}) \\ &= 0.95\text{ns} + 0.135\text{ns} + 0.1\text{ns} \\ &= 1.185\text{ns}\end{aligned}$$

注意到 1.185ns 的平均延迟近乎相当于 1ns 延迟，这是为存储器层级中最快的存储器 L1 cache 所作的假定。

10.2 cache 映射

在层级结构中从主存开始的每一个低层都比其上一层存储器含有更多的块。因此，每个缓冲存储器都必须实现一个方法，以便快速验证被请求的块的副本是否在 cache 里或该副本是否需要被传输到低一层的存储器。由于 cache 只是一个简单的快速临时存储空间，CPU 发出的主存地址被划分为块地址和偏移值，这能够标识出块内特定的字节/字。接下来，块地址则用于指定块在 cache 中的位置，即槽地址（或槽号）或索引。

例 10-2 考虑 64KB 的主存，1KB 的 L2 cache 和 8B 的块。找出存储器块的数目、
[432] cache 槽的数目、块地址的范围和槽地址的范围。

解：块的数量通过按照块的大小划分主存大小得到，如下所示：

$$\begin{aligned}\text{主存中块的数目 } N &= 64\text{KB} / (8\text{B/block}) \\ &= 2^{16}/2^3 \\ &= 2^{13} \\ &= 8192 \text{ 块}\end{aligned}$$

cache 中槽的数目由相似方法得到：

$$\begin{aligned}\text{cache 槽的数目 } K &= 1\text{KB} / (8\text{B/slot}) \\ &= 2^{10}/2^3 \\ &= 2^7 \\ &= 128 \text{ 个槽}\end{aligned}$$

如图 10-3 所示，16 位（ $2^{16}\text{B} = 64\text{KB}$ ）的主存地址被划分为 13 位的块地址和 3 位的偏移值。块地址的范围是 0 ~ 8191，槽地址的范围是 0 ~ 127。注意，缓冲存储器只能保存主存中所有块的 1.56%（ $128/8192 * 100$ ）。

直接映射和组相联映射是将块地址映射为槽地址所普遍采用的两种方法，它们适用于在层级结构中从主存开始的任意两个相连的存储器之间。在直接映射的 cache 中，一个块的副本只能存储在一个特定的 cache 槽中。在组相联映射的 cache 中，一个副本可以存到小型槽集合的某个槽中。直接映射 cache 更简单、更快、效能更高，因为一个副本只能映射到一个槽中。

在一些应用程序中，cache 缺失会造成大得多的延迟，全相联映射经常被用于此。在这种情况下，数据能被存到任意的 cache 槽中，从而达到更高的命中率。然而，全相联映射 cache 需要更多硬件并且所有槽可以并行（同时）查询。在接下来的几节里将会详细讨论直接映射和组相联映射 cache 组织。在虚拟内存系统的设计中对全相联映射 cache 的应用将会在 10.4 节讲述。

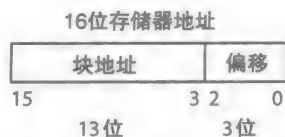


图 10-3 内存地址由块地址和偏移值组成，每个块假定为 8B

10.2.1 直接映射

对于直接映射 cache，我们需要两个信息来快速确定 cache 中是否包含目标块的副本。一个信息是通过计算模 (Mod) 值得到的槽地址，另一个是通过整除法得到的，叫作标记。433
例如，考虑 129 和 1153 这两个块地址以及一个含有 $K = 128$ 个槽的缓冲存储器。对于这两个块地址，相应的槽和标记值计算如下：

$$\begin{aligned} 129 \bmod 128 &\Rightarrow \text{槽 } 1 \\ 129/128 &\Rightarrow \text{标记 } 1 \\ 1153 \bmod 128 &\Rightarrow \text{槽 } 1 \\ 1153/128 &\Rightarrow \text{标记 } 9 \end{aligned}$$

然而，如果 K 是 2 的幂 (例如， $K = 2^m$)，映射很简单而且无需硬件，如公式 (10-1) 和公式 (10-2) 用 n 位块地址 X 所示。

$$\begin{aligned} \text{槽地址} &= X \bmod 2^m \\ &= (x_{n-1}x_{n-1}\cdots x_mx_{m-1}\cdots x_0)_2 \bmod 2^m \end{aligned} \tag{10-1}$$

$$\begin{aligned} &= (x_{m-1}\cdots x_0)_2 = > \text{低 } m \text{ 位} = \text{槽} \\ \text{标记} &= X/K \\ &= (x_{n-1}x_{n-1}\cdots x_mx_{m-1}\cdots x_0)_2/2^m \end{aligned} \tag{10-2}$$

进一步的推导如下所示，其中块地址 $X = 16$ 和 29 ，使用的 $K = 8 (2^3)$ ：

$$\begin{aligned} 16 \bmod 8 &= (10000)_2 \bmod 2^3 = 0 && \text{槽} = (000)_2 \text{ 最低 } 3 \text{ 位} \\ 29 \bmod 8 &= (11101)_2 \bmod 2^3 = 5 && \text{槽} = (101)_2 \text{ 最低 } 3 \text{ 位} \\ 16/8 &= (10000)_2/8 = 2 && \text{标记} = (10)_2 \text{ 最高 } 2 \text{ 位} \\ 29 \bmod 8 &= (11101)_2/8 = 3 && \text{标记} = (11)_2 \text{ 最高 } 2 \text{ 位} \end{aligned}$$

cache 组织

表 10-1 展示了两个主存地址 0x408 和 0x240B，分别指向块 129 的 0 字节和块 1153 的 3 字节。如图 10-4 所示，只有一个块 (用地址值表示) 的副本能存到槽 1 中。也存储在 cache 中的标记表示包含了副本的槽中的块地址 (129 或 1153)。在图中，标记 = $9 = (1001)_2$ 表示槽 1 包含块 1153 的副本。

表 10-1 使用 64KB 主存、1KB L2 cache 和 8B 块的直接映射 cache 示例

| 内存地址 | | 块地址 (十进制) | 标记 (十进制) | 槽地址 (十进制) | 偏移值 (十进制) | 目标字节 |
|-----------|----------------------|--------------|-------------|--------------|--------------|------------|
| 地址 (十六进制) | 划分为标记、槽和偏移值的地址 | | | | | |
| 408 | 000001, 0000001, 000 | 129 | 1 | 1 | 0 | Slot 1: B0 |
| 240B | 001001, 0000001, 011 | 1153 | 9 | 1 | 3 | Slot 1: B3 |

434

cache 中每个槽还存储了额外的表明块副本状态的位。这些位被用来实现 cache 一致性协议，以保证陈旧数据不被送到 CPU 或存到硬盘。这些位如图中的 cache 一致性位 (CCB) 所示。

图 10-5 说明了在图 10-4 中逻辑显示的直接映射 cache 的数据通路。数据通路由 1 个 128 项的标记存储器和一个 128 项的行存储器组成。cache 的容量最大能存储 128 个块。使用槽地址能同时访问到标记存储器和行存储器。传入的标记会与标记存储器中存储的标记进行比较。如果两个标记匹配，则 cache 访问命中，否则为缺失。出于性能考虑，当传入的标

记与 cache 中存储的标记比较时，由多路复用器找出目标字节（或字）。如图所示，若 cache 访问命中，该字节在一个读周期内被送到 CPU（假设只有一个 L1 cache）。若访问为缺失，则 cache 控制器被触发以便从与之相连的低层存储器里访问该块。程序的执行会暂停，直到 cache 缺失得到解决。

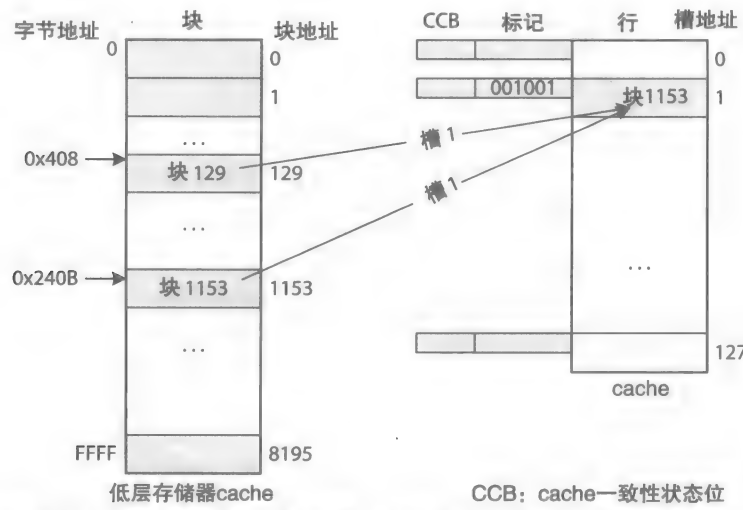


图 10-4 一个图示了块 129 和 1153 映射到槽 1 的直接映射 cache 的逻辑视图，块 1153 复制到槽 1 的过程也予以了展示。内存地址 0x408 和 0x240B 指向每个块中的一个字节

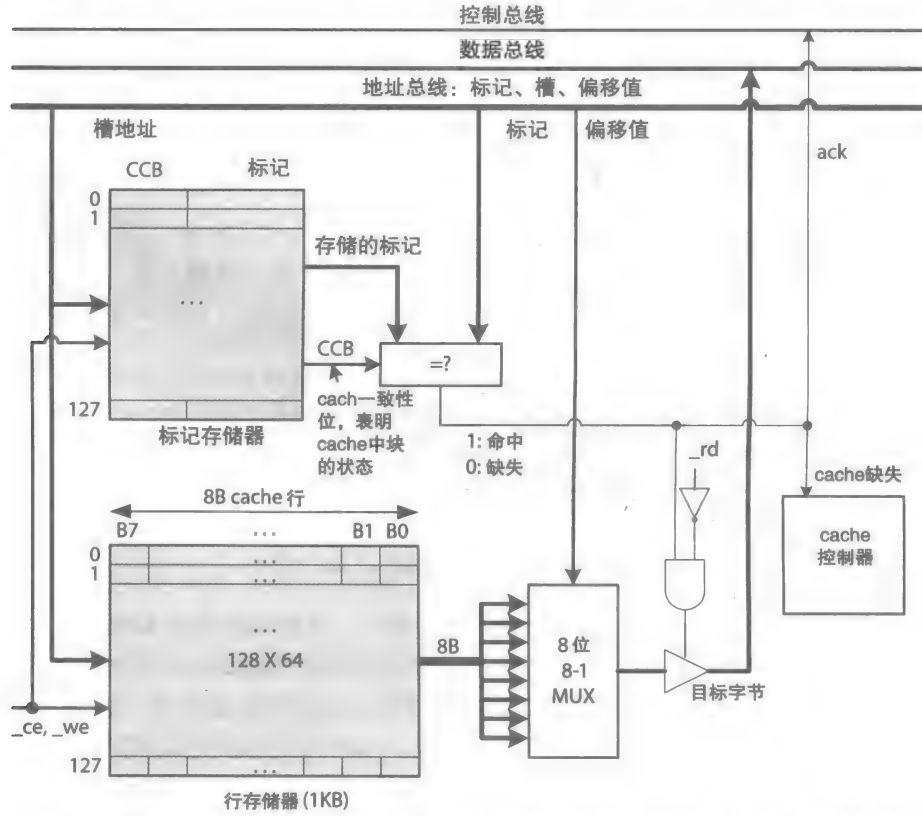


图 10-5 用 128 个槽和 8B 块说明 cache 读取过程的直接映射 cache 数据通路

10.2.2 cache 缺失的类型

大体上，cache 缺失分为冷缺失、冲突缺失、容量缺失、真共享缺失和伪共享缺失。下面的例子说明了冷缺失、冲突缺失和容量缺失。真和伪共享缺失与两个或多个线程访问共享数据块有关，这将在 10.3 节讨论。

435

例 10-3 考虑图 10-4 所示的直接映射 cache。假定一个 L1 cache，设定 CPU 按序访问下面的内存地址 20 次。求出这个内存访问序列的 cache 缺失率。另外，求出冷缺失、容量缺失和冲突缺失的次数。地址都是 16 位的，按十六进制给出。

- A. 0x3C10 假定地址指向内存块 Ba 中的一个字节
- B. 0x049C 假定地址指向内存块 Bb 中的一个字节
- C. 0x0410 假定地址指向内存块 Bc 中的一个字节
- D. 0x1C8D 假定地址指向内存块 Bd 中的一个字节

解：我们首先求出 cache 缺失的数量，然后用来求解该内存访问序列的缺失率。从块地址到槽地址的映射由表 10-2 给出。在第一轮中，4 个内存地址造成了 4 次 cache 缺失。这些与块 Ba、Bb 和 Bd 关联的缺失是冷缺失，因为这些块被复制到 cache 中的三个初始为空的槽。地址 C 也映射到与地址 A 相同的槽 2。然而，由于这些地址的标记不同，它们指向主存中两个不同的块中的字节。因此，Bc 的副本代替了槽 2 中 Ba 的副本，导致一次冲突缺失。这称为冲突缺失，因为 cache 中仍然存在空的槽，但是 Ba 和 Bc 的副本必须存储到同一个槽 2 中。在第一轮，总共有 3 次冷缺失和 1 次冲突缺失。

表 10-2 在例 10-3 中给定的 4 个地址的直接映射

| 第一次被 CPU 访问 | | | | | | |
|-------------|----------------------|------------------------|---------|----------|--------|------------|
| # | 地址：标记，行号，偏移（二进制） | | 标记（十进制） | 行地址（十进制） | 字节 | 命中 / 缺失 |
| A | 001111, 0000010, 000 | | 15 | 2 | B0 | M |
| B | 000001, 0010011, 100 | | 1 | 19 | B4 | M |
| C | 000001, 0000010, 000 | | 1 | 2 | B0 | M |
| D | 000111, 0010001, 101 | | 7 | 17 | B5 | M |
| # | 第二次访问 | | | 第三次访问 | | 第 4 到 20 轮 |
| A | M | 行 2：冲突，拷贝 Ba 覆盖 Bc 的拷贝 | | M | 与第二轮相同 | M |
| B | H | 拷贝 Bb 到行 19 | | H | 与第二轮相同 | H |
| C | M | 行 2：冲突，拷贝 Bc 覆盖 Ba 的拷贝 | | M | 与第二轮相同 | M |
| D | H | 拷贝 Bd 到行 17 | | H | 与第二轮相同 | H |

在第二轮，块 Bb 和 Bd 的副本已经在 cache 中，任何对这些副本进行的读 / 写都会导致 cache 命中。当再次访问地址 A 时，在第一轮被 Bc 的副本替换掉的 Ba 的副本又会替换掉块 Bc 的副本，这导致又一次的冲突缺失。再次访问地址 C 又导致另一次的冲突缺失，并且用 Bc 的副本替换掉槽 2 中 Ba 的副本。在第二轮，不存在冷缺失，但有两次冲突缺失。

第二轮中命中 / 缺失的模式会在剩下的 18 轮里重复进行，每一轮都会得到 0 次冷缺失和两次冲突缺失。这 4 个地址不会造成任何容量缺失，因为 cache 中仍然存在可用的空槽。如公式（10-3）的求解，对这 4 个地址访问 20 次，总共会造成 42 次 cache 缺失，得到的缺失率是 52.5%。

$$\begin{aligned} \text{缺失的总次数} &= 3 \text{ 次冷缺失} + 1 \text{ 次冲突缺失} + (20 - 1) \text{ 轮} * 2 \text{ 次冲突缺失 / 轮} \\ &= 42 \text{ 次总缺失} \end{aligned}$$

$$\text{缺失率} = \text{总缺失数} / \text{总访问数}$$

$$= 42 / (20 * 4) = 42 / 80 = 0.525$$

(10-3)

例 10-4 考虑图 10-4 所示的直接映射 cache。假定从 0 ~ 255 的主存块被按序访问 10 次。这些块的内存地址为 0x0 到 0x7FF。求出该内存访问序列的 cache 缺失率。另外也求出冷缺失、容量缺失和冲突缺失的次数。

解：同样，我们首先求解 cache 缺失的次数，然后用它求出该内存访问序列的缺失率。从块地址到槽地址的映射由表 10-3 给出。在第一轮，最先的 128 个块（0 ~ 127）充满了整个 cache，每一个都会导致一次冷缺失，总共 128 次冷缺失，这是因为槽初始都是空的。接下来的 128 个块（128 ~ 255）中的每个块都会导致一次缺失并且替换掉已存在于 cache 中的块 0 到块 127 的一个副本。例如，块 128 的副本会替换掉块 0 的副本，块 129 的副本会替换掉块 1 的副本，等等。

表 10-3 在例 10-4 中给定的主存地址的直接映射

| 地址（十六进制） | 地址（二进制） | 块号（十进制） | 标记（十进制） | 行号（十进制） | 命中 / 缺失 |
|----------|----------------------|---------|---------|---------|---------|
| 0x0 | 000000, 0000000, 000 | 0 | 0 | 0 | M |
| 0x8 | 000000, 0000001, 000 | 1 | 0 | 1 | M |
| 0x10 | 000000, 0000010, 000 | 2 | 0 | 2 | M |
| ... | ... | ... | ... | ... | M |
| 0x3F8 | 000000, 1111111, 000 | 127 | 0 | 127 | M |
| 0x400 | 000001, 0000000, 000 | 128 | 1 | 0 | M |
| 0x408 | 000001, 0000001, 000 | 129 | 1 | 1 | M |
| 0x410 | 000001, 0000010, 000 | 130 | 1 | 2 | M |
| ... | ... | ... | ... | ... | M |
| 0x7F8 | 000001, 1111111, 000 | 255 | 1 | 127 | M |

因此，在第一轮，访问块 128 到块 255 会造成 128 次容量缺失（原因稍后讲述）。在第二轮，块 0 ~ 127 的副本在第一轮已被块 128 ~ 255 替换出来了，每一个块都会导致一次容量缺失。现在，块 0 的副本将会替换掉 cache 中块 128 的副本，块 1 的副本将会替换掉块 129 的副本，等等。在第二轮中对块 128 ~ 255 的访问将会替换掉已存在于 cache 中的块 0 ~ 127 的副本，从而导致第二轮总共 256 次容量缺失。

第二轮中的容量缺失模式将会在剩下的 8 轮中一直重复。在奇数轮，块 0 ~ 127 的副本会替换掉 cache 中块 128 ~ 255 的副本，而在偶数轮（从 2 开始），块 128 ~ 255 的副本会替换掉块 0 ~ 127 的副本，导致每轮都是 256 次容量缺失。如公式（10-4）所得，10 轮里总的 cache 缺失次数为 2560，缺失率为 100%。在这个案例中没有冲突缺失。

$$\begin{aligned} \text{缺失总次数} &= 128 \text{ 次冷缺失} + 128 \text{ 次容量缺失} + (10 - 1) \text{ 轮} * 256 \text{ 次容量缺失 / 轮} \\ &= 128 \text{ 次冷缺失} + 2432 \text{ 次容量缺失} \\ &= 2560 \text{ 次总缺失} \end{aligned}$$

(10-4)

$$\begin{aligned} \text{缺失率} &= 2560 \text{ 次缺失} / (10 * 256) \text{ 次访问} \\ &= 1.0 \end{aligned}$$

容量缺失与不依赖于 cache 大小的冲突缺失差别明显。例如，如果例 10-4 中的 cache 大

小从 1KB 增加到 2KB, cache 将有足够的空间容纳所有 256 个块的副本; 因此, 这会导致仅在第一轮有 256 次冷缺失而在剩下的 9 轮里面没有缺失。这将会改善 100% 的缺失率到 10% ($256/2560$), 而命中率从 0% 变为 90%。一般说来, cache 的大小是固定的, 并在设计之时就被确定下来。然而, 缓存的仿真可以用来选择一个合适的 cache 大小, 从而不会导致太多的容量缺失。冲突缺失和容量缺失有时是组合在一起的。

通常, 缺失率由块大小决定。例如, 正如所预期的, 块越小, 冷缺失会越多。使用更大的块型号会减少冷缺失的数量。然而, 块的大小和缺失率的关系是程序依赖, 但是通常来说, 当与 cache 大小相关的块太小或太大都会增大缺失率。

10.2.3 组相联映射

直接映射很简单并且只需要较少的硬件, 但是它的缺点是太受限制了。如例 10-3 所示, 映射到同一个槽地址的两个或多个频繁引用的块地址可能产生频繁的 cache 缺失并且导致延迟。图 10-6 就展示了一个带有这样的两个块的程序, 它们在内存中相距 1KB。假定 cache 为 1KB 并且块大小为 8B。如图所示, 块 129 包含一个短的例行程序的指令, 块 1153 包含一个 for 循环的指令。当 for 循环重复执行和短例行程序被重复调用时, 都直接映射到槽 1 (表 10-1) 的块 129 和块 1153 将会在 cache 中重复地互相替换副本。这将会增加执行时间, 因为 I_c 必须频繁地从 L2 cache 中取回块 129 和 1153 的指令。这也增加了 CPU 空闲时间; 然而, 多线程 (第 8 章) 可以使这种 CPU 空闲时间最小化。

组相联映射是一种减小由于冲突造成的缺失的方法, 例如图 10-6 所示的内存访问情景。在组相联映射中, 缓冲存储器被组织成组, 每一个组都有少量 (如 2、3、4 或 8) 的槽。块地址直接映射成组地址, 而不是像在直接映射中所做的映射成槽地址。在每个组内, 一个块能被存储到某个槽中, 这是由替换算法决定的, 例如近期最少使用法 (LRU)、轮询法 (循环形式, 先进先出) 或随机法。

替换算法由硬件实现, 这可能使组相联 cache 更复杂和更慢。与其他两种替换算法相比, LRU 需要最多的硬件, 而随机法需要最少的硬件。已经发现的是, 在 cache 型号很小的时候, 就低 cache 缺失率而言, LRU 通常是最好的, 而随机法是最差的。当 cache 型号较大时, LRU 和随机法的性能相近并且都好于轮询法 [1]。更多关于实现复杂度的讨论请参考练习部分。

图 10-7 展示了块地址到一个二路 (两个组) 1KB 的组相联 cache, 假定块大小为 8B。128 个 cache 槽被分为 $64 = 2^6$ 个组, 每个组有两个槽。通过使用公式 (10-1) 和公式 (10-2) 并替换 $m = 6$, 块地址被转换成一个组地址和一个标记。表 10-4 给出了两个存储器地址 0x408 和 0x240B 到一个 64 组的二路组相联 cache 中的映射。

如图 10-7 所示, 块 1153 先被复制到 cache (假定为图 10-6 所示的内存访问情景) 并存

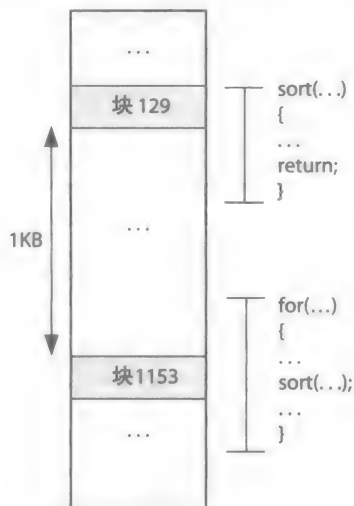
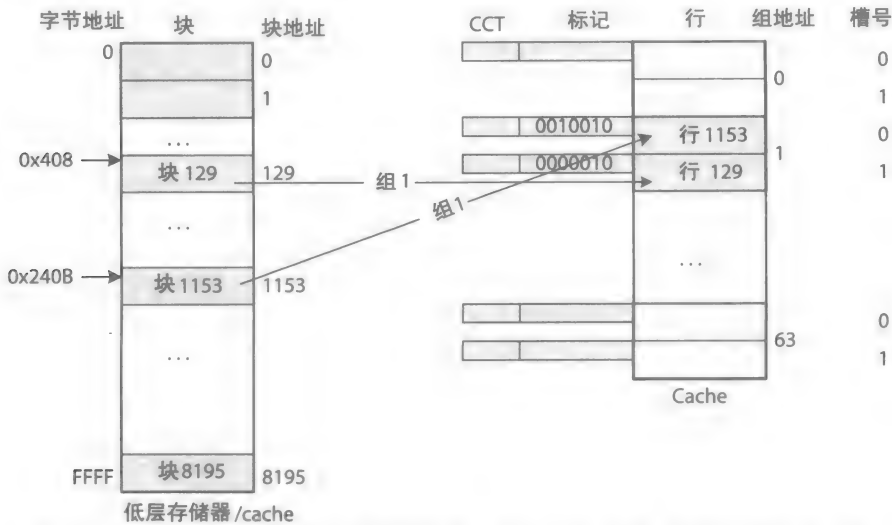


图 10-6 一个展示了直接映射的 cache 的局限性的程序示例; cache 为 1KB, 块大小为 8B

储在组 1 的槽 0 中。块 129 也映射到组 1，它的副本现在存储到组 1 的槽 1 中。因此，与使用图 10-4 中的直接映射 cache 造成的很多次缺失相比，图 10-6 中程序的执行在对块 129 和 1153 访问时只会造成两次 cache 缺失。



440 图 10-7 二路组相联 cache 的逻辑视图，共有 128 个槽，每两个槽分为一组

表 10-4 块 129 和 1153 到一个 64 组二路组相联 cache 的映射

| 目标字节地址 | | 块地址 (十进制) | 标记 (十进制) | 组号 (十进制) | 偏移值 (十进制) | 目标字节 |
|-----------|----------------------|--------------|-------------|-------------|--------------|---------|
| 地址 (十六进制) | 地址: 标记、组, 偏移值 (二进制) | | | | | |
| 408 | 0000010, 000001, 000 | 129 | 2 | 1 | 0 | 行 1: B0 |
| 240B | 0010010, 000001, 011 | 1153 | 18 | 1 | 3 | 行 0: B3 |

cache 组织结构

图 10-8 展示了如图 10-7 中逻辑显示的二路组相联 cache 的数据通路。顶部的标记和行存储器是为 64 个组中每个槽 0 保留的。底部的标记和行存储器则是为 64 个组中每个槽 1 保留的。在一次 cache 读 / 写周期中，所有的 4 个存储器会被同时访问，并且到来的标记会与存储在两个标记存储器中的标记都进行比较。如果目标块的副本在 cache 中，从标记存储器中读到的两个标记之一会与到来的标记匹配成功，从而导致一次 cache 命中。反之，副本不在 cache 中并且该次访问会导致一次 cache 缺失。

由于在二路组相联 cache 中所有的 4 个存储器模块会被同时访问，组相联 cache 会消耗更多能耗。一种降低能耗的方法是使用路预测组相联 cache，只有一对标记 - 行存储器会被首先查询。如果这会导致一次 cache 缺失，则接下来所有成对的标记 - 行存储器会被同时查询 [2]。

在组相联 cache 中路数不必是 2 的倍数。例如，三路组相联 cache 将会需要三对标记、行存储器模块；在图 10-8 中只有两对。英特尔的八核 Xeon 处理器包含了一个 24MB 的共享 L3 cache，它被组织成一个 8 端口 3 路（称为 24 路）组相联 cache。只要访问是由 8 个不同的组发起的，相连的 8 个 L2 cache 就能同时访问到 L3 cache。

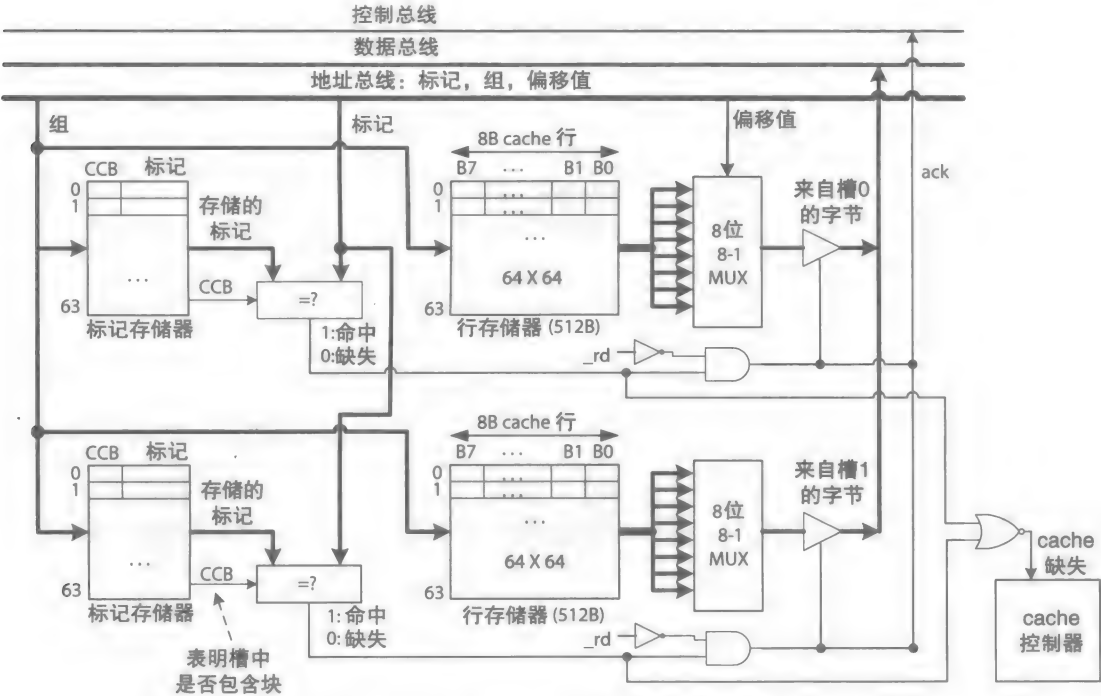


图 10-8 一个 64 组二路组相联 cache 中表明一次 cache 读命中的数据通路

例 10-5 考虑如图 10-7 逻辑展示的组相联 cache。假设 CPU 按顺序引用了例 10-3 给定的地址 20 次。求出这个内存访问序列的 cache 缺失率。同时，求出冷缺失、容量缺失和冲突缺失的次数。

解：表 10-5 表示了将块地址映射到组地址的计算结果。在第一轮中，这些地址产生了 4 次冷缺失。块 Bb 和 Bd 的副本被装载到两个不同组的两个空槽中。块 Ba 和 Bc 的地址都映射到组 2，但是这次 Ba 和 Bc 的副本被存储到组 2 的两个不同的槽中。在剩下的 19 轮中，由于这 4 个块的副本都在 cache 中，4 个地址不会导致任何一次 cache 缺失，也没有容量缺失。如下面的计算所示，现在总的缺失次数为 4，缺失率改善到了 5%（命中率 = 95%）。这与使用直接映射 cache（见例 10-3）造成的 42 次总缺失和缺失率 = 52.5%（命中率 = 47.5%）形成对比。

缺失的总次数 = 4 次冷缺失 + (20 - 1) * 0 次缺失 / 轮

= 4 次总缺失

命中率 = 4 次缺失 / (20 * 4) 次访问

= 0.05

表 10-5 例 10-3 所给地址的二路组相联映射

| 第一轮 | | | | | | | |
|-----|-----------------------|----------|-----------|----|----|---------|-----------------|
| # | 地址: 标记, 行号, 偏移值 (二进制) | 标记 (十进制) | 组地址 (十进制) | 行号 | 字节 | 命中 / 缺失 | 注释 |
| A | 0011110, 000010, 000 | 30 | 2 | 0 | B0 | M | 拷贝 Ba 到组 2 行 0 |
| B | 0000010, 010011, 100 | 2 | 19 | 0 | B4 | M | 拷贝 Bb 到组 19 行 0 |
| C | 0000010, 000010, 000 | 2 | 2 | 1 | B0 | M | 拷贝 Bc 到组 2 行 1 |
| D | 0001110, 010001, 101 | 7 | 17 | 0 | B5 | M | 拷贝 Bd 到组 17 行 0 |

(续)

| # | 第 2 轮 | | | 第 3 轮 | | 第 4 轮到 20 轮 | |
|---|-------|-----------------|--|-------|--------|-------------|--------|
| A | H | 拷贝 Ba 到组 2 行 0 | | H | 与第二轮相同 | H | 与第二轮相同 |
| B | H | 拷贝 Bb 到组 19 行 0 | | H | 与第二轮相同 | H | 与第二轮相同 |
| C | H | 拷贝 Bc 到组 2 行 1 | | H | 与第二轮相同 | H | 与第二轮相同 |
| D | H | 拷贝 Bd 到组 17 行 0 | | H | 与第二轮相同 | H | 与第二轮相同 |

10.3 cache 一致性

每一个 cache 存储器必须实行一致性协议从而保证每次读周期返回的数据为该数据块的最新副本，无论最新副本是在 cache 中还是在主存储器中。例如，考虑图 10-9 中的双处理器系统。为了简单起见，每个处理器只显示了一个核心和一个与存储器总线相连的 L2 cache。该系统还包含一个桥来连接存储器总线和 I/O 总线。DMA 控制器在主存储器和磁盘驱动器之间传输页数据，图中还显示了两个存储器块的两个副本 Ba 和 Bb，其已复制到 cache 存储器中。

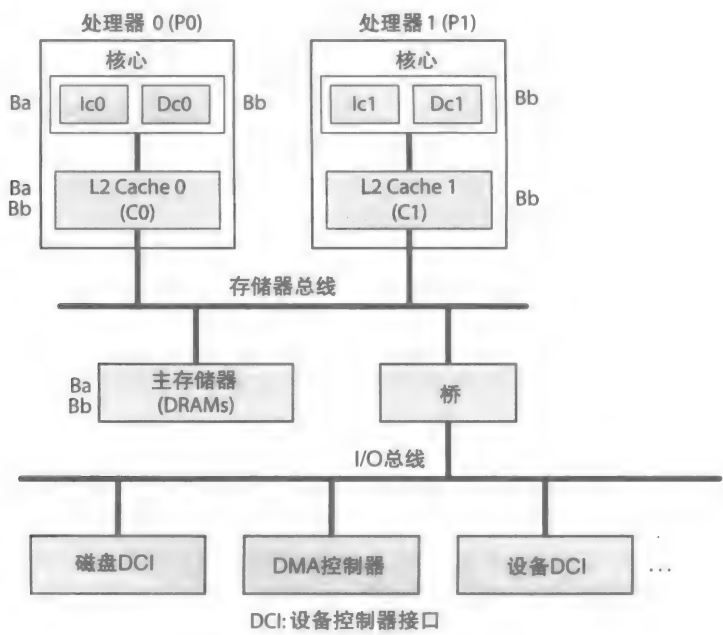


图 10-9 双处理器 UMA 架构

图中，处理器 1 (P1) 可能执行一个写回指令从而更新 Dc1 中块 Bb 的副本中的某个字。因为 Bb 的副本也存在其他 cache 和主存储器中，直到其他 cache 和主存储器意识到本次更新，在此中的副本将保持原样，与 DC 中的不同。相似的，若 DMA 控制器向主存储器中写一块，如 Ba，Ba 的副本也在其他一个或多个 cache 中，这也将导致主存储器中和 cache 中的 Ba 不同。

因为 CPU 轮询完成写周期，在其他 cache 中的块副本 (如果有) 是无效的或是已被更新的。这两个选项称为失效和更新高速 cache 一致性协议。一个混合 cache 实行失效一致性协

441
2
443

议和更新一致性协议的组合。

10.3.1 失效协议与更新协议

在一个失效协议中,当某个 cache 中的一个块进行了更新,其他 cache 中的该块的副本都将失效,从而阻止这些过期副本被访问。包含无效副本的 cache 存储器,当其需要使用该副本时,其需要请求来获取最新副本。无效协议的劣势在于,即使某块中只有一个字进行的更新,整个块都会被置为无效。这将增加 cache 缺失的概率,尤其是当处理器(或内核)访问一个共享块时。

例如,假设图 10-9 中的处理器 P0 访问了块 Bb 的前半部分,P1 访问了 Bb 的后半部分。每当 P0 向其 Bb 块的副本写数据时,在 C1 和 Dc1 中的 Bb 的副本变为无效。如果 P1 再次访问 Bb 块,在 Dc1 中将会产生一次 cache 缺失,即使处理器 0 只更新了块 Bb 副本的前半部分,并未修改 P1 访问的 Bb 副本后半部分,Dc1 仍旧必须先请求更新 Bb 的副本。在这种情况下,cache 缺失也被称为假共享,因为 P0 和 P1 并未真正共享块 Ba 中的数据。如果处理器 P0 和 P1 的确访问了块 Ba 中的相同的一个或多个数据项,Dc1 上的 cache 缺失被称为真共享缺失。

在一个更新协议中,每个 cache 存储器必须要其他 cache 广播并告知任何更新。当其他 cache 被告知时,其必须更新其中副本的内容(如果有)。然而,更新协议的劣势在于,因为增加了不必要的更新操作导致整体总线通信状况较差。例如,假设 P1 对块 Bb 中的数据进行了处理,但是 Bb 也在 C1 中,每当 P0 向 Bb 写数据时,更新协议要求其必须告知 C1 对 Bb 的改动,即使 P1 不再需要访问块 Bb。

因为更新协议会潜在地增加协议相关的通信传输,此协议并不常用。然而,混合 cache 协议可以使用一种自适应模式来利用两种协议的优点。例如,混合 cache 协议将在某块被更新一定次数后会置其他 cache 中该块为无效。在这种情况下,如果 P1 正在处理块 Ba,而 C1 中含有块 Ba 的副本,C1 中的块 Ba 将被置为无效,并阻止进一步不必要的更新。下面我们将讨论两种常用的无效协议写模式:直写,写回。

10.3.2 监听 cache 一致性协议

每一个 cache 存储器需要两个控制器:cache 控制器和监听控制器。cache 控制器用于响应来自高等级 cache 的请求,当控制器属于一个指令 cache 或数据 cache (Ic 或 Dc) 时,cache 用于响应来自处理器核心的请求。监听控制器用于响应来自较低等级 cache 的请求,当控制器属于最低等级 cache 时,其也用于响应出现在存储器总线上的读/写事务。

例如,在图 10-9 中,Dc0 的 cache 控制器用于响应来自 P0 的请求。Dc0 的监听控制器用于响应来自 cacheC0 的请求。相似的,C0 的 cache 控制器响应来自 Ic0 或 Dc0 的请求,C0 的监听控制器用于响应存储器总线上的事务。

假设采用无效协议,当图 10-9 中的 DMA 控制器向存储块 Ba 发起了一个写事务时,连续监听存储器总线的 C0 监听控制器检测到写动作,并置其中 Ba 的副本为无效。继而,C0 将与更高等级的 cache Ic0 或 Dc0 进行通信并使 Ba 的副本为无效(如果有)。相似的,由两个 L2 cache 中的一个发起的存储器写事务将被另一个 L2 cache 的监听控制器检测到,在其处理器中的该块的副本(如果有)将置为无效。cache 控制器和监听控制器实行更新协议的方式与之前相似,唯一不同的是 cache 中的副本将用新值更新而不是之前置为

无效。

10.3.3 直写协议

直写协议是一种写无效协议。正如其名所示，处理器处理的所有写请求将穿过 cache 直接更新主存储器。当一次写命中时（cache 命中是由写周期造成的），不仅 cache 中该块的副本将被更新，存储器中该块的副本也将被更新。然而，如果 cache 中没有该块的副本，写周期将更新主存储器，但该块的副本将不被加载到 cache 中。这么做的原因是，cache 写事务将转发数据到主存储器中，无论数据的副本是否在 cache 中，都没有必要将该块的副本复制到 cache 中。这也被称为无分配直写 cache 协议，如图 10-10 中有限状态图（FSD）所示。

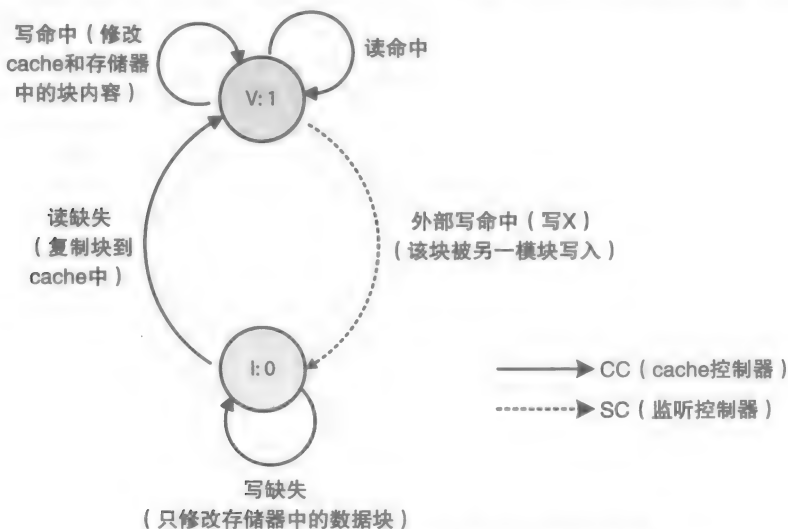


图 10-10 说明无分配直写协议的有限状态图

直写 cache 在每块中使用 1 位来表示该块的状态为合法（V）或不合法（I），如 FSD 中所示。状态 V 和状态 I 被编码为 1 位 CCB（图 10-5）并存储在标记存储器中。例如，1 可以用于表示 cache 中该副本为合法，0 表示为非法或未加载到 cache 中。

FSD 说明了 5 种可能的转换，为 $I_{WM} \rightarrow I$ 、 $I_{RM} \rightarrow V$ 、 $V_{WH} \rightarrow V$ 、 $V_{RH} \rightarrow V$ 、 $V_{XWH} \rightarrow I$ 。WM、RM、WH、RH 和 XWH 分别表示写缺失、读缺失、写命中、读命中和外部写命中（在其他 cache 中的写命中）。例如，正如前面关于写命中所说明的那样，写缺失是由写周期造成的 cache 缺失，而读缺失是由读周期造成的。

直写协议的优势在于其较为简单，在每个 cache 中，一块的副本只能为两种状态中的一种。然而，这种协议的劣势是其可能造成总线过载和 cache 堵塞，故而，在多核或多处理器系统中不推荐使用直写协议。

例如，考虑图 10-9 中的多处理器系统，并假设 cache 使用图 10-10 的直写协议。假定 P0 中运行下列代码段。因为 *sum* 被声明为全局变量，数组中的元素之和将不被存储在寄存器中，相反，每当数组中的下一个元素加到部分和时，在主存储器和 cache 中包含 *sum* 的该块都将被更新。因为在 for 循环中，*sum* 被更新了 100 次，Dc0 需要发送 100 个存储器事务来更新 L2 cache，相应的，L2 cache 将发送 100 个写事务来更新主存储器，造成了存储器带宽的浪费。


```
int sum = 0; //global variable sum
main()
{
    int array[100]; //locally declared array
    . . .
    for(i = 0; i< 100; i++)
    {
        sum = sum + array[i];
    }
    . . .
}
```

10.3.4 写回协议

写回协议被设计用来在保持 cache 一致性的前提下降低不必要的总线使用和 cache 过载，常用的写 - 回无效协议为 MESI（发音为 “messy”）协议，此外，还有其他两种 MESI 类型协议，MESIF（用于 Intel）和 MOESI（用于 AMD）被设计用来提升 cache-cache 间的通信效率。

446

1. MESI

在 MESI 协议中，存储器数据块的 cache 拷贝可以为以下 4 种状态之一：修改（M）、专有（E）、共享（S）、无效（I）或不存在，MESI 协议的 FSD 如图 10-11 所示。状态 M 表示 cache 中该块的拷贝已经被修改（脏的），不再是“干净的”——干净指的是其与主存储器中的拷贝相同。状态 E 表示 cache 中含有主存储器之外该块的唯一拷贝。状态 S 表示有 2 个或多个 cache 含有该块的拷贝，此外，每份拷贝都是“干净的”。

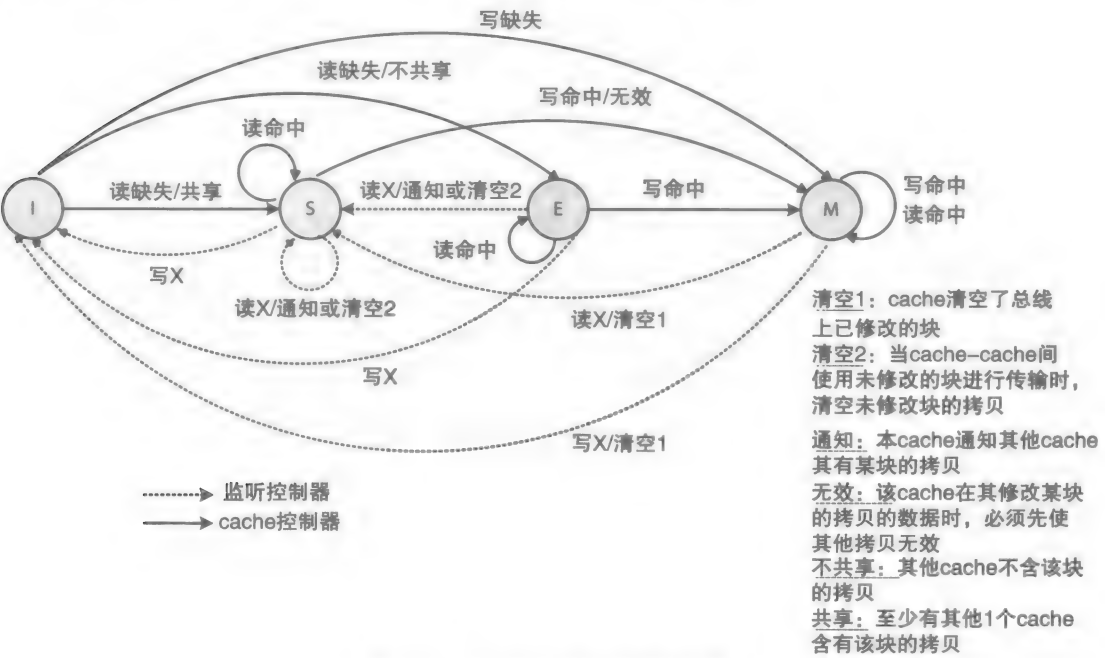


图 10-11 MESI 协议有限状态图

表 10-6 描述了 MESI 中各种状态转移。MESI 协议相对于直写协议而言，极大地降低了通信流量，但是其也比直写协议复杂得多。例如，为了更新共享的拷贝（S），MESI cache 必

须执行以下任务：

表 10-6 MESI 协议中的状态事务

| 状态转移 | 描 述 |
|-------------------------|--|
| $I_{RM} \rightarrow S$ | 其他 cache 中也含有该块拷贝时发生读缺失 |
| $I_{RM} \rightarrow E$ | 没有其他 cache 含有该块拷贝时发生读缺失 |
| $I_{WM} \rightarrow M$ | 写缺失 |
| $S_{RH} \rightarrow S$ | 对一个共享块的读命中 |
| $S_{WH} \rightarrow M$ | 对一个共享块的写命中，在修改该数据块的拷贝之前，必须告知其他 cache 其对该块进行了修改 |
| $E_{WH} \rightarrow M$ | 对一个专有块的写命中，cache 不需要先通知其他 cache 即可进行修改操作 |
| $E_{RH} \rightarrow E$ | 对一个专有块的读命中 |
| $M_{RH} \rightarrow M$ | 对一个已经修改块的读命中 |
| $M_{WH} \rightarrow M$ | 对一个已经修改块的写命中 |
| $S_{XWH} \rightarrow I$ | 外部写命中，有其他 cache 请求修改该块 |
| $M_{XRH} \rightarrow S$ | 外部读命中，其他 cache 请求获取该块的拷贝，cache 必须先清空总线上的该块（清空 1），使得其他 cache 可以加载该块 |
| $E_{XWH} \rightarrow I$ | 外部写命中，其他 cache 正在向该块写入数据 |
| $M_{XWH} \rightarrow I$ | 外部写命中，其他 cache 请求写该已修改的块，本 cache 必须清空（清空 1）总线上该修改块并置其为无效 |
| $S_{XRM} \rightarrow S$ | 外部读缺失，其他 cache 正在从主存储器或其他 cache（清空 2）中加载该块的拷贝 |
| $E_{XRH} \rightarrow S$ | 外部读命中，其他 cache 正在从主存储器或另一个 cache（清空 2）中加载该块的拷贝 |

RM：读缺失，WM：写缺失，RH：读命中，WH：写命中，X：外部的

1) cache 在修改 S 状态的拷贝前，必须通知其他 cache（通过监听协议）。这将确保其他 cache 中该块的拷贝为无效。

2) cache 之后将修改该块的拷贝，并改变该块拷贝的状态由 S 变为 M。同时，其监听控制器将负责响应来自其他 cache 或 DMA 控制器对该块的读 / 写请求。

3) 如果拷贝被替换，cache 必须向存储器写入该已修改的块拷贝。

然而，对一个 E 状态的块拷贝的写命中并不需要监听控制器告知其他 cache，因此，MESI 协议减少了总线事务并降低了 cache 通信流量。

在当今多数采用 MESI 协议的计算机中，通常是主存储器而不是 cache 负责发送 S 状态的块拷贝给请求拷贝的 cache。此外，当从一个 cache 向一个请求 cache 中传输一个已修改的数据块时，主存储器中该块的拷贝也将被更新，使得在 cache 和主存储器中所有的块拷贝都变为干净的，转为状态 S。

在一个 NUMA 系统中，所有的存储器空间被分割为多个结点，如图 10-12 所示。NUMA 系统中每一个结点都包含一个用于内部结点相互通信的通信接口（CI），一个请求结点的 CI 将向目的结点的 CI 按一定的路径发送一个远程存储器事务，从而在两个结点间生成一个虚拟连接。例如，在图中，CI0 和 CI1 间的通信将使得结点 0 和结点 1 中的存储器总线出现连接，从而在结点 0 和结点 1 间出现无缝通信。然而，当多个 cache 请求访问一个共享块时，这种点对点的通信方式将潜在的增加通信延迟。例如，考虑图中的块 Ba，若该 cache 采用无“清空 2”选项的 MESI 协议（如图 10-11 中读 X/ 清空 2 的弧线），M3 将需要发送 Ba 块的拷贝给所有请求获得该块的 cache。这将潜在地使 M3 成为一个需引入延时的热点，因此增加了平均延时。相似的，如果 cache 采用含有“清空 2”选项的 MESI 协议来

447
448

进行 cache-cache 间的通信, 含有共享拷贝的处理器将变为一个热点。两种先前引入的协议 MESIF 和 MOSEI 将用于解决这个问题, 我们将在接下来讨论。

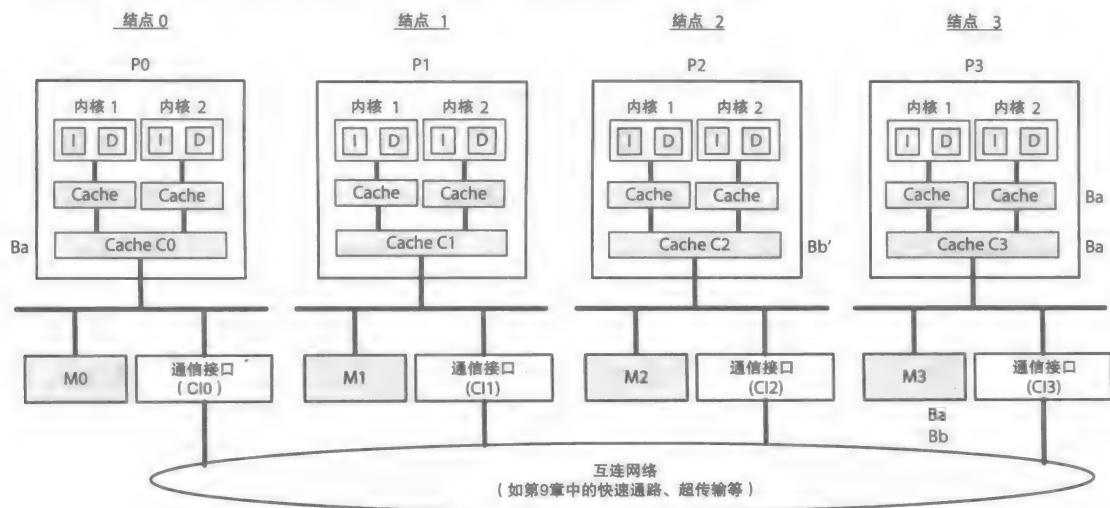


图 10-12 3 个结点 NUMA 系统框图

2. MESIF

MESIF 协议设计用来实现 cache-cache 间的共享数据块（S）的通信。当两个或多个 cache 含有共享数据块的拷贝时, 其中一个（第一个）cache 中的拷贝将标记为 F（转发），其他 cache 中该块的拷贝将标记为 S。包含状态 F 的数据块的 cache 负责传输（转发）该数据块的拷贝给下一个接收拷贝的 cache。然而, 在传输之后, 传输的 cache 将改变其数据块的状态从 F 到 S, 接收的 cache 将存储拷贝并将该拷贝状态改为 F。

因为只有一个 cache 中含有状态为 F 的数据块拷贝, 该数据块在进行一次传输后状态将置为 S, MESIF 协议从而阻止了任一 cache 成为一个热点, 通过标记 F 状态使得 cache 轮流传输共享块的拷贝给请求的 cache。注意, 在图 10-12 中的 NUMA 系统中, 因为每个结点只包含一个处理器, MESIF 将开启结点和结点间的对共享拷贝的通信。如果每个结点为 UMA 多处理器系统, cache-cache 间的通信允许在结点中的每个 cache 从本地 cache 中获取共享数据块拷贝（如果有），而不是从另一个结点中获取, 从而避免了不必要的延迟。MESIF 协议中 M、E、S、I 状态工作方式和 MESI 协议中相同。

例如, 假设图 10-12 中的 4 结点 NUMA 系统采用 MESIF 协议, 并且每个结点含有一个目录（未在图中显示）, 在结点的每个存储块中, 该目录包含一系列对该数据块拷贝进行了 cache 的结点名称。此外, 假设图 10-12 中的块 Ba 已载入 cache C3（结点 3）, 并已发送一份拷贝给 cache C0（结点 0）。在结点 3 中关于块 Ba 的目录列出了结点 0 和结点 3, 并标记结点 0 为 F。另一个 cache, 如 C1（结点 1）, 对 Ba 块的请求将首先发送给结点 3。CI3 通信接口将检查关于块 Ba 的目录并转发 C1 的请求给结点 0, 再改变目录使其包含结点 1, 但此时其将标记结点 1 为 F。结点 3 下次接收到关于状态位 S 的 Ba 拷贝的请求时, 在结点 1 中的包含状态为 F 的数据块的 cache 将负责发送该拷贝。在目录中, 一个包含已修改拷贝（如 Bb', 其中 ' 表示其已修改）的结点（如结点 2）将被标记为 M。

449

与 MESI 协议相似, 在 MESIF 中, 每当一个已修改拷贝传输到另一个处理器时, 主存

存储器中该拷贝也将更新。在一个 NUMA 结构中, 这要求包含已修改拷贝的 cache 发送二次事务从而更新相应的存储器单元。注意, 在一个 cache 采用 MESI 协议的 UMA 结构中, 其中最低等级的 cache 和主存储器共享一个公用总线, 当总线上检测到传输一个已修改数据块时, 主存储器中的内容也将更新 (通过其监听控制器)。

3. MOESI

与 MESIF 相反, MOESI 协议设计用来支持 cache-cache 间对已修改拷贝的通信。使用状态 O 来表示拥有, 并被用来在不更新主存储器的情况下共享已修改数据块的拷贝。当某个 cache 请求获取一个已修改数据块的拷贝时, 含有该已修改拷贝的 cache 将传输该拷贝给请求的 cache, 并改变其拷贝的状态由 M 到 S。然而, 接收的 cache 将设置其拷贝状态为 O。下一次发生对该已修改拷贝的请求时, 含有状态 O 拷贝的 cache 将负责传输拷贝到请求的 cache 中。在一次传输后, 源 cache 中该拷贝的状态将由 O 变为 S, 而目的 cache 中该拷贝的状态将为 O。因此, 和 MESIF 协议一样, 采用 MOESI 协议的系统阻止了一个 cache 变为热点。

注意, 在 MOESI 协议中, 对干净的拷贝的请求必须来自于相应的存储器单元, 除非一个 M 或 O 状态的拷贝存在于另一个 cache 中。在 MOESI 协议中, 对状态为 M 或 O 的拷贝的更换要求一次事务来更新存储器。相似的, 在一个采用 MOESI 协议的 NUMA 系统中, 每个结点中的目录中将记录一系列含有 S 状态拷贝的 cache 名, 其中只有一个标记为状态 O (如果有)。注意, 在 MOESI 协议中, S 状态的拷贝与主存储器中的拷贝可能相同也可能不相同。记录包含已修改拷贝的结点的目录与 MESIF 协议中的相同。

10.4 虚拟存储器

现代单核、多核和多处理器计算机系统采用多道程序设计, 使得多个单线程或多线程的程序 (第 8 章) 并行执行。也就是说, 操作系统轮转给每个线程分配一定的 CPU 时间供其执行。在一个含有多个处理器核心的多核处理器或多处理器系统中, 多线程将并行执行。如果每个核心还采用同步多线程技术 (第 8 章), 更多数量的线程可并行执行。

[450]

OS 为了可分配 CPU 时间给各个线程, 其需要给每个正在运行的单线程或多线程程序分配主存储器空间, 这一程序被称为进程。虽然单个进程可能不共享其分配的内存空间, 但多线程程序的线程可共享分配给该进程的内存空间 (如: 一个程序中的多个线程均可访问该程序中声明的全局变量)。因此, 现代计算机系统必须实现关于存储器等级中对底层两级 (非易失性存储器和主存储器) 的下列要求:

- 可运行一个对于物理存储器而言更大的程序。程序包含太多的指令和大数据结构以致于其不可被完整地存储到主存储器中。
- 当主存储器中没有足够的空间来同时存储所有进程的指令和数据时, 操作系统依然可以执行多进程。
- 保护进程使得一个进程在未被允许时不可访问另一进程的存储器空间。

现代计算机系统采用虚拟存储器来实现以上三个要求。当一个程序运行时, 必须从非易失性存储器 (如硬盘) 中拷贝该程序相关的指令和数据到主存储器中, 以供执行。然而, 因为物理存储器的大小比硬盘要小得多, 例如 cache, 每个进程存储在硬盘上的指令和数据只有一小部分存储到物理存储器的可用空间中。

例如, 一个地址总线宽度为 32 位、数据总线宽度为 32 位的 32 位 CPU 最多可读写 4GB (2^{32} B) 存储器空间, 组织结构为 $2^{30} \times 32$ 的存储器单元。即使物理存储器空间如此之

大，也无法装载包括操作系统相关进程的所有进程的指令和数据。

因此，32 位 CPU 可访问的全部 4GB 存储空间被解释为虚拟空间而不是实际的物理空间。此外，为了可以同时运行操作系统和用户进程，4GB 虚拟空间中的一半（2GB）将被保留以供用户进程使用，另外一半供系统进程使用。在 CPU 状态寄存器中，使用单独的一位来表示 CPU 在用户模式且地址属于用户进程或者 CPU 处于内核模式且地址属于系统进程。分配给各个进程的虚拟存储器空间将进一步划分为多个指令域和数据域（见第 8 章中图 8-5）。然而，一个非常大的（大小大于 4GB）程序需要在 64 位计算机系统中编译运行。

当一个操作系统给每个线程分配一定的 CPU 时间，操作系统每次被定时器中断时，将执行一次上下文切换（第 9 章）。在一个上下文切换过程中，刚刚执行一片 CPU 时间的线程的状态将被保存，而等待执行的线程的状态将被恢复，使得每个线程的执行过程和无上下文切换时的执行过程是相同的，从而给用户产生每个线程一个 CPU 的错觉。

[451]

如果切换没有改变 CPU 当前访问的虚拟地址空间，上下文切换也被称为**线程切换**，否则将称为**进程切换**。进程切换后，CPU 将从一个新的虚拟空间执行一个线程。在本节剩余部分，我们将主要关注与存储器相关的进程切换。

和从物理存储器拷贝指令和数据块到 cache 中相似，指令和数据块，每个称为一页，当需要时，从虚拟存储器（例如硬盘）拷贝到物理存储器（参考第 9 章中的 DMA 传输）。如果一个物理页的内容是脏的（修改过的），在用新的虚拟页内容代替其之前，必须将其中的内容拷贝回硬盘中。

在现代 PC 系统中，每一页的大小一般为 4KB（一个相对较大的块），这个大小使得在硬盘和物理存储器之间的 DMA 传输效率更高。如果存储器空间划分为更大尺寸的页，对于一个进程，将产生更少的页缺失，和使用更大尺寸块时 cache 冷缺失情况更少相似。某些系统可能使用可变大小的页，称之为段，但是此处，我们主要关注基于页的虚拟存储器管理系统。

10.4.1 虚拟地址转换

图 10-13 使用包含两个进程的系统状态的用例说明了基于页的虚拟存储器管理系统。在图中，进程 0 包含 12 个虚拟页，而进程 1 包含 9 个虚拟页，物理存储器包含 8 个物理页。在一个进程执行过程中，当需要时，存储器系统必须映射并存储进程虚拟页中的内容到物理存储器中。映射是全相联的，因此，虚拟页中的内容可存储到物理存储器中的任何一页。然而，不同于存储器-cache 映射，存储器-cache 映射全部由硬件完成，虚拟-物理的存储器地址映射部分由软件（如 OS）执行部分由硬件模块执行，该硬件模块被称为内存管理单元（MMU）。

在图 10-13 中，对于进程 0 虚拟页标号为 0 ~ 11，进程 1 标号为 0 ~ 8。进程 0 中的虚拟页 3、5、8 分别映射到物理页 5、2、7。进程 1 中的虚拟页 2、5 分别映射到物理页 3、0。物理页 1、4、6 未被占用，为可用的。

虚拟存储器系统使用页表来保存虚拟-物理的页映射记录。例如，当使用大小为 4KB 的页，2GB 的虚拟存储器需要 512K（2GB/4KB）条目录的页表。因为每个进程含有其自身的页表，当系统中进程数量提升时，页表的数量也将提升。因此，一般而言，一些最不被访问的表项可能会暂时存储在硬盘上，到需要时复制到物理存储器中。

[452]

图 10-14 概述了进程 0 中虚拟页 3、5、8 到物理页 5、2、7 的映射。在图中，假设每一页大小为 256B，每个虚拟存储空间为 32KB（ 2^{15} B），包含 128（32KB/256B）个虚拟页，物理存储空间为 8KB，包含 32（8KB/256B）个物理页。在这种情况下，15 位的虚拟地址可看

作包含 7 位的虚拟页号 (VPN, 高 7 位) 和 8 位的页偏移 (低 8 位)。页偏移与 cache 块偏移相似, 用于在一个虚拟页或物理页中确定目标字节或目标字。

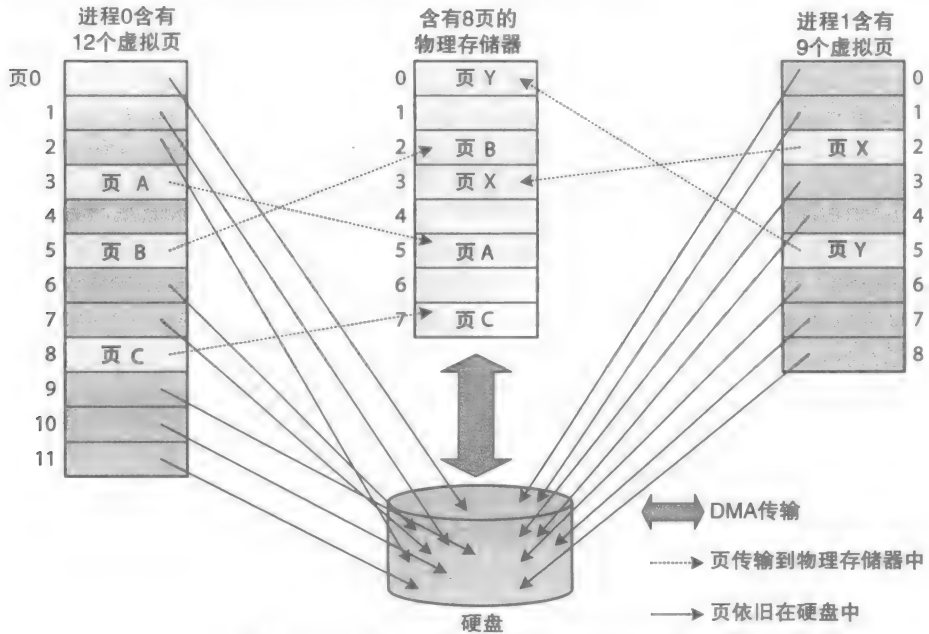


图 10-13 虚拟-物理的存储器页映射图

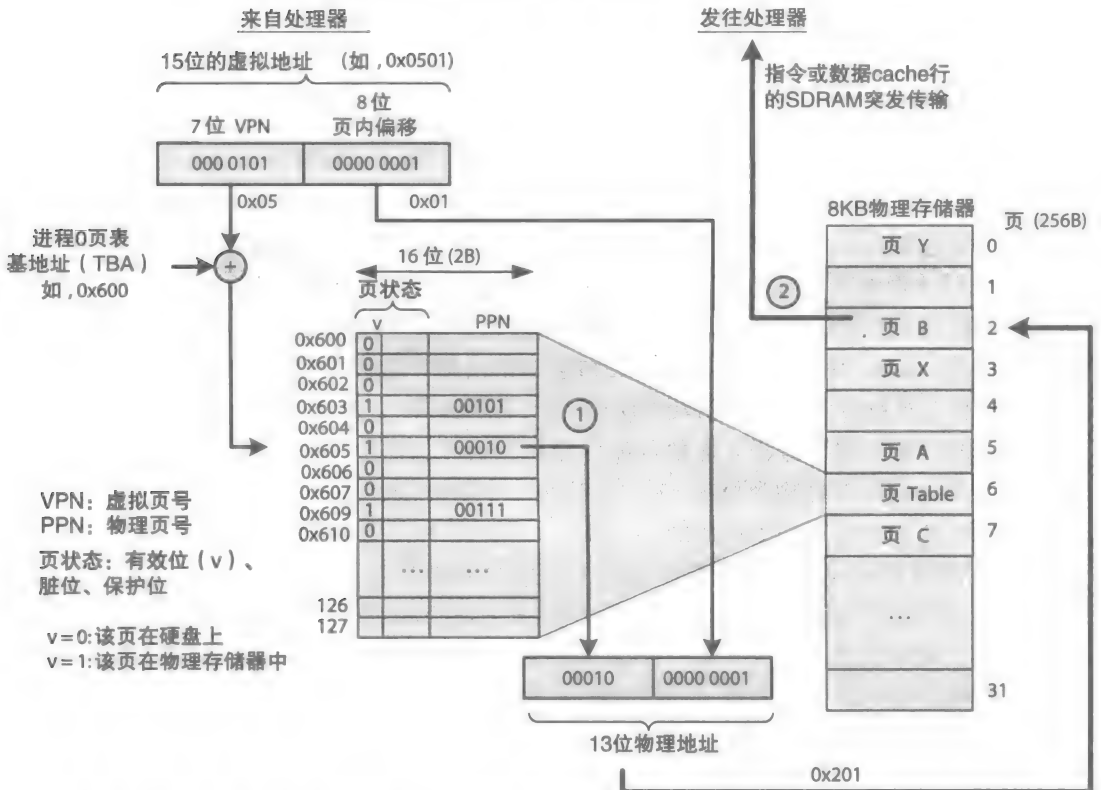


图 10-14 图 10-13 中进程 0 中的虚拟页地址通过 MMU 的虚拟-物理地址转换步骤

假设每个 128- 页表中每条记录 16 位 (2B)。每条记录包含 5 位 ($2^5 = 32$) 的物理页地址 (PPN) 和一系列状态位, 如有效位 (v) 和访存控制位 (如读、写、脏、用户和监听等)。当有效位为 1 时表明表中该记录包含了一条有效的 PPN。

特殊的, 当采用虚拟存储器系统时, 程序从虚拟地址 0 开始执行, 包括指令地址和数据地址——例如, 在执行 LD 指令和 ST 指令 (第 8 章) 过程中——都采用虚拟地址。采用图 10-14 中说明, 以下步骤描述了 MMU 转换 15 位虚拟地址 0x0501 到 13 位物理地址 0x0201 和物理存储器转换相应的 cache 行到处理器的操作过程:

1) MMU 将 15 位的虚拟地址 0x0501 看作 7 位的虚拟页号 $VPN = 5$ (0x05) 和 8 位的页偏移 0x01。使用 $VPN = 5$ 作为索引, MMU 将访问进程 0 的页表, 其页表存储在物理存储器中页表基地址 $TBA = 0x600$ 处, 如图所示。在页表中, 与 $VPN = 5$ 相对应的记录包含 $PPN = 2 = (00010)_2$ 及有效位 $v = 1$, 这表明对应的物理页号为 2 并且有效 (包含了最新的数据)。物理页号 2 再与页内偏移 0x01 一起生成一个 13 位的有效物理存储器地址 0x0201。

2) 接下来, 物理存储器将传输包含地址 0x0201 的数据块作为一个 cache 行到处理器中, 如图所示。

因此, 处理器接收虚拟地址 0x0501 中的内容需要两次独立的物理存储器访问。在第一次访问中, MMU 直接访问物理存储器转换虚拟地址 0x0501 为物理地址 0x0201。在第二次访问中, 物理存储器将响应 cache 缺失, 传输包含物理地址 0x0201 内容的数据块给处理器。

在系统中, 因为虚拟存储器空间非常大, 其需要一个非常大的页表, MMU 在完成虚拟-物理地址转换之前将需要一定时间来访问物理存储器。多级页表是一种组织存储超大页表的解决方式, 在多级页表中的每条记录, 当其不处于最低级时, 其还包含一个供查询下一级页表的 TBA, 最低等级的页表将包含 PPN。

假设页大小为 256B 的系统含有 4MB (2^{22} B) 的虚拟地址空间。共有 16K 个虚拟页 (4MB/256B), 远远多于图 10-14 中的仅包含 128 个虚拟页的系统。图 10-15 说明了采用 2 级页表结构组织存储 16K 虚拟页的方式。其中, MMU 将虚拟地址中 22 位地址看作 3 部分, 如图 10-15 所示: 7 位的索引用于访问一级页表 (图中页表 1), 7 位的索引用于访问二级页表 (图中页表 2) 和 8 位的页内偏移 (最低 8 位)。

MMU 使用最高 7 位地址 0x00 作为索引来从一级页表中读取二级页表的地址 $TAB = 0x400$, 用于访问一级页表的 TBA 为 0x600, 如图 10-15 所示。接着, MMU 将使用第二部分 7 位地址从二级页表 (最低等级页表) 中获取相应的物理页地址 $PPN = 2 = (00010)_2$ 以及有效位 $v = 1$ 。PPN = 2 再与 8 位的页内偏移地址组合成目标 13 位物理地址。如上所示, 在这种情况下, 相对于图 10-14 只需要一次访问物理存储器, 其需要两次访问物理存储器来转换虚拟地址 (如 0x0501) 为相应的物理地址 (如 0x0201)。

10.4.2 转译后备缓冲器

为了降低虚拟-物理地址转换的长延迟, 最常引用的 PPN 也保存在一个专用的全相联 cache 存储器中, 这也被称为转译后备缓冲器 (TLB), 通常称为快表。图 10-16 说明了含有 32 个槽的全相联 TLB 组织结构。与采用直接映射和组相联映射的 cache 中所使用的标记和数据存储不同, 全相联 TLB 中需要使用寄存器来存储标记和 PPN。其不需要行号和组地址, 采用并行的方式搜索目标标记, 如图 10-16 所示。

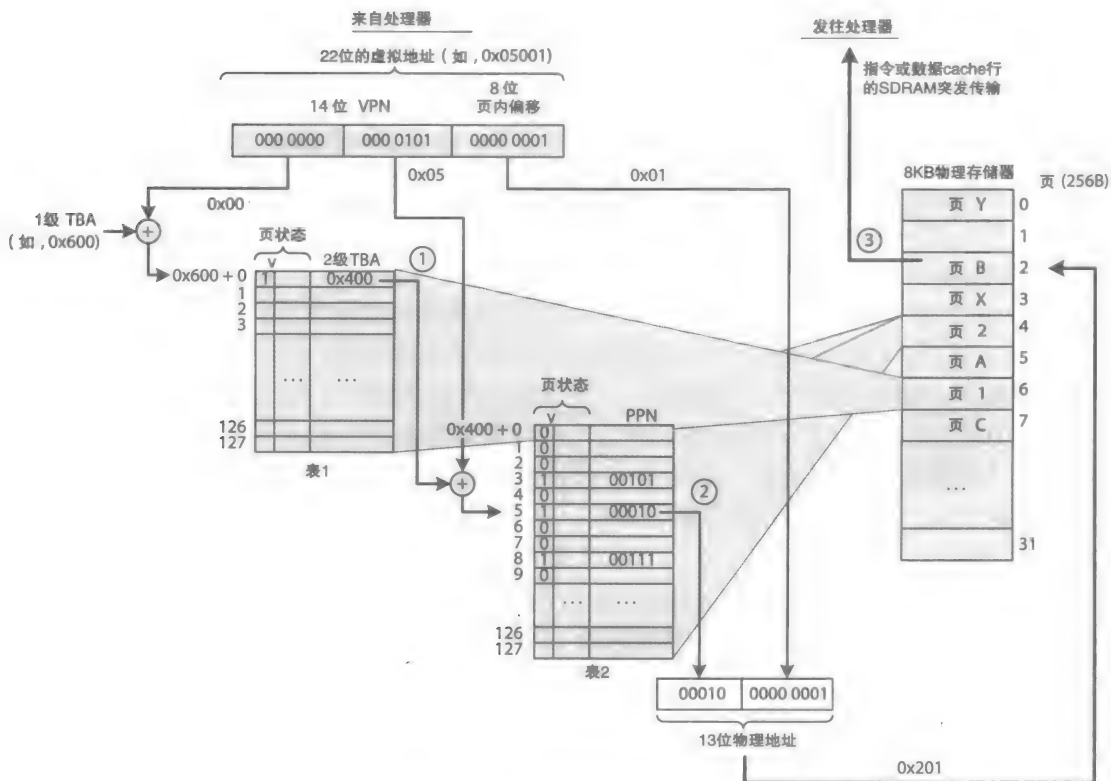


图 10-15 2级页表组织结构

TLB 并不需要 cache 一致性标志位, 如图所示, 其保存了来自页表的状态位, 包括脏位 (d)。一开始, 处理器第一次访问一个数据页时, 在 TLB 中该页的 PPN 标记为不脏 ($d=0$)。如果处理器写 (直写或写回) 并修改了来自该页的某个数据块, 在 TLB 中该记录的 d 位变为 1。当 $d=1$ 时, 其表明在物理存储器中该页已修改, 在其被一个新转换的虚拟页替换之前, 需将该页内容写回到硬盘上。并且, 由于 TLB 是全相联的, 其必须实行一种槽替换算法。

TLB 的替换算法相对于 LRU 而言, 需使用更少的硬件, 如图所示, 其在每个目标寄存器中使用一个使用位 (u) 来记录该记录使用。每当从 TLB 中访问一个 PPN 时, 该 PPN 对应的记录中 u 位将置为 1。其他所有记录的 u 位将置复位为 0, 表示这些记录最近未访问。每当 TLB 访问产生一次缺失时, TLB 将开始一个新的虚拟-物理地址转换, 并替换一条 $u=0$ 的记录槽。如果被替换的记录中 $d=1$ (表示该页在物理存储器中已修改), 在页表中该已修改页的 d 位也将置为 1, 该槽继而使用刚确定的 PPN 进行更新。

10.4.3 处理器组织结构

图 10-17 说明了 3 种处理器内部组织结构。在图 10-17a 中, 使用两个 TLB 快速转换两个虚拟地址, 一个用于指令, 一个用于数据 (如果有), 并在 L1 cache 使用各自的物理地址之前, 传输相应的物理地址给 L1 cache。这种结构的优势在于其在 L1 cache 中使用物理地址寻址 cache, 而图 10-17b 中使用虚拟地址寻址 cache。其劣势在于增加了 L1 cache 传输时延。

图 10-17b 中的组织结构使用虚拟地址寻址的 cache, 其使用一个单独的 TLB 在对该物

理存储器的访问之前快速转换来自最低等级 cache（本例中为 L2 cache）的虚拟地址为相应的物理地址并传输给物理存储器。然而，这种组织结构并不常用，其在每次进程切换时要求 OS 来清空所有的 cache。如果没有清空，cache 无法辨别不同进程同一虚拟地址，例如图 10-13 中进程 0 中 $VPN = 5$ 和进程 1 中 $VPN = 5$ 。这种组织结构的优势在于其不会增加 L1cache 的时延。

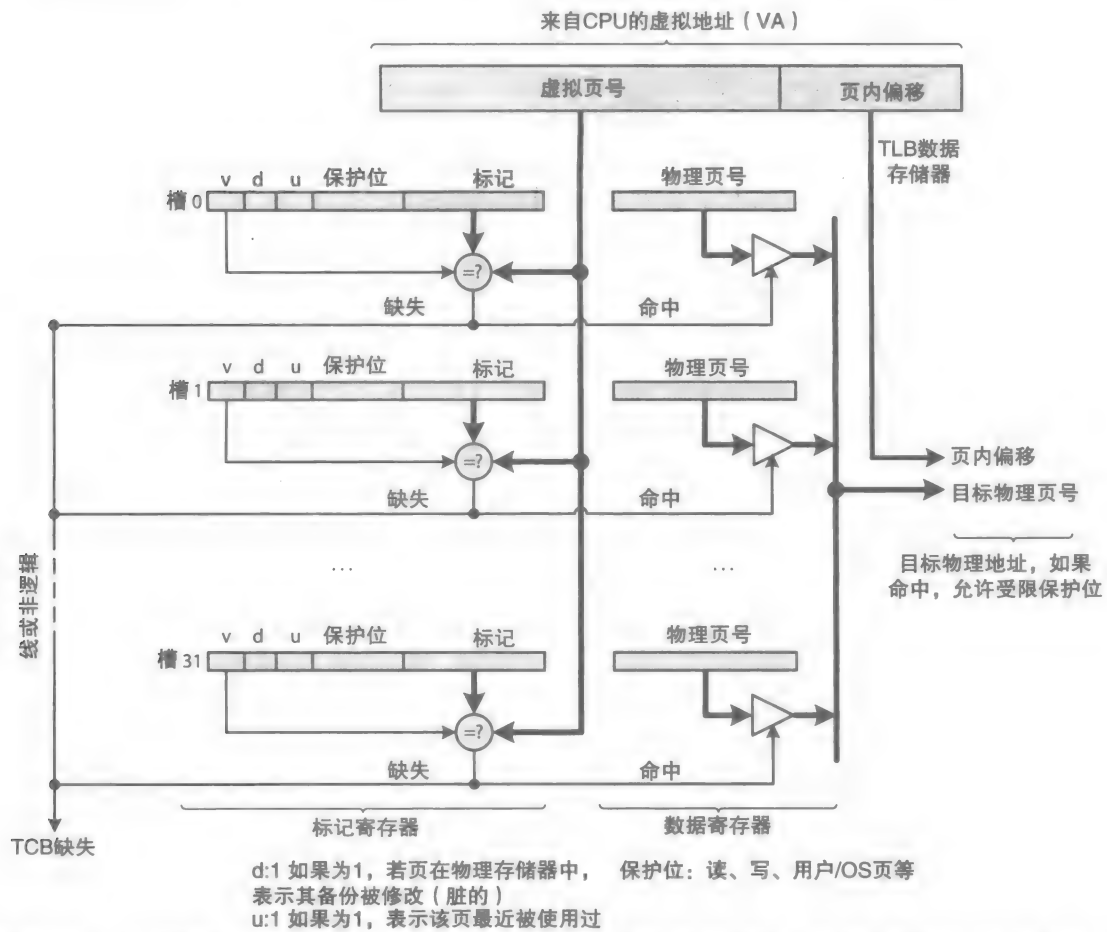


图 10-16 全相联 TLB 数据通路, 说明 TLB 读过程 (未显示所有细节), 缺失将导致 MMU 转换一个虚拟地址到物理地址

在图 10-17c 的组织结构中, 在每个 L1 cache 中嵌入了一个 TLB, 如图 10-18 所示, 使用一个标记存储器存储 PPN 并以 PPN 为标记。在一次 cache 访问过程中, 命中或是缺失是通过比较由 TLB 得来的 PPN 和从标记存储器中获取的 PPN 来确定的。这个 cache 被称为虚拟地址寻址但使用物理地址标记的, 如 AMD Opteron 处理器中使用的 cache 采用这种组织结构。

这种组织结构有两大优势: 1) 其使用物理地址寻址的 cache, 2) 其 L1 cache 的延迟最小, 与图 10-17b 中相同。然而, 这种组织结构的劣势在于每个 L1 cache 的大小必须小于或等于一页的大小。例如, 如果页大小为 4KB, L1 cache 最大大小为 4KB。使用这种组织结构的高性能处理器需要采用大尺寸页的虚拟存储器系统。

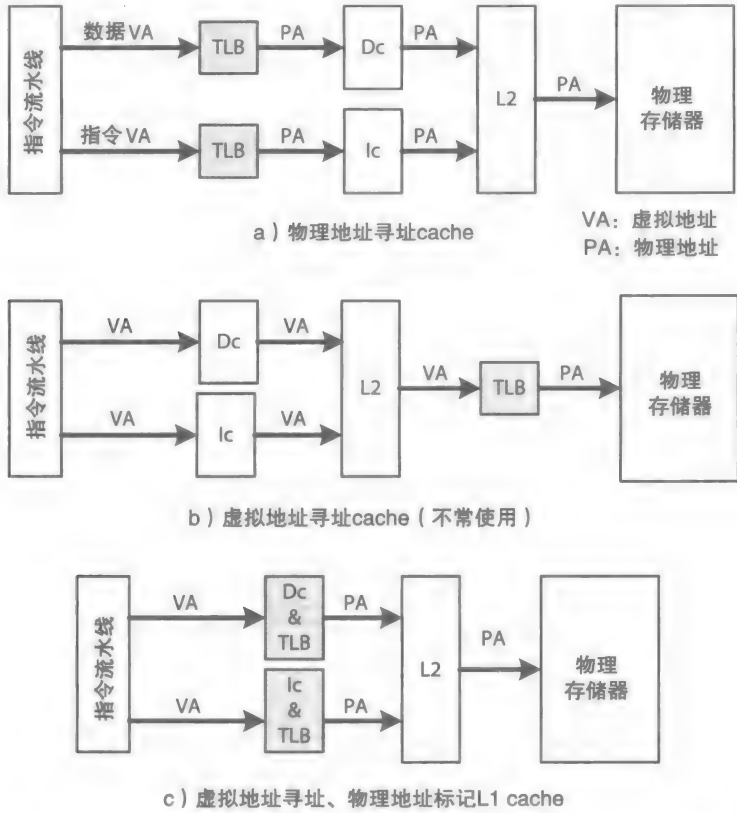


图 10-17 不同的处理器组织结构: a) L1 cache 延迟较长, b) L1 cache 延迟较短, 但 OS 在每次进程切换时必须清空 cache, c) L1 cache 延时较短, 但 TLB 的大小必须比页大小小

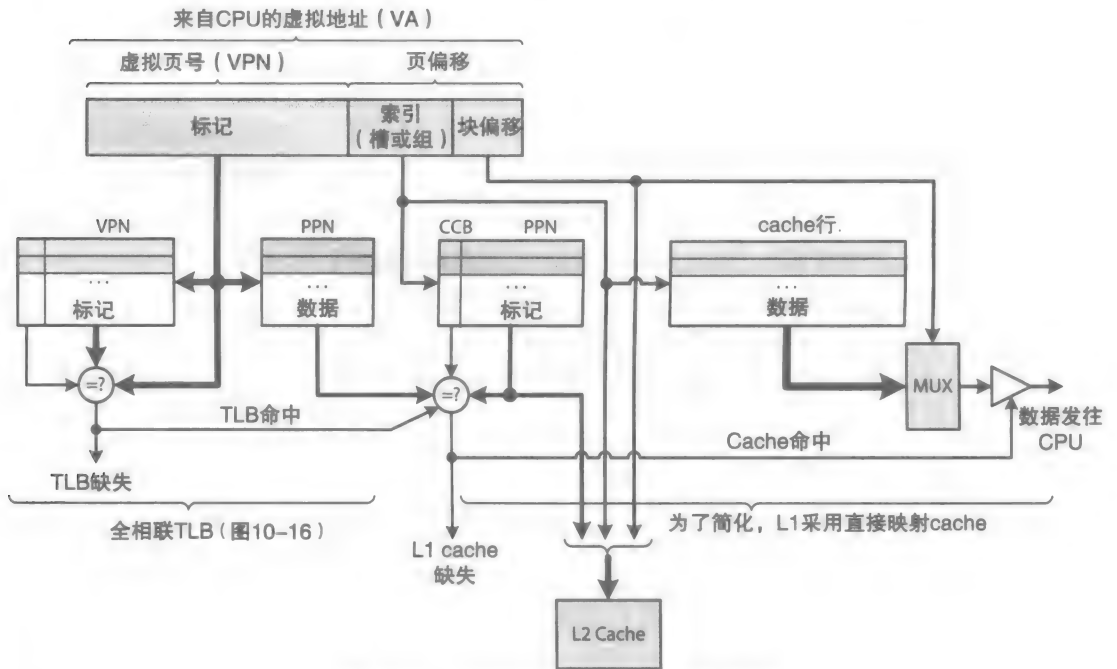


图 10-18 cache 中嵌入 TLB 组织结构

此外，对于图 10-17a 和图 10-17c 中的 TLB，处理器也可以包含一个 2 级（L2）TLB，和 L2 cache 相似。L2 级的 TLB 可存储更大量级的 PPN。如果在一级 TLB 中搜索一个 PPN 产生一个一次未命中，将在 L2 TLB 中继续搜索。如果搜索还是未命中，MMU 将触发一个目标 VPN 到相应的 PPN 的转换。一般情况下，一个 L2 TLB 要远远大于 L1 TLB，其采用直接映射或组相联映射的 cache。

参考文献

1. John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 5th ed., Waltham, 2012.
2. Inoue Koji, Ishihara Tohru, Murakami Kazuaki, Way-predicting set-associative cache for high performance and low energy consumption, *ACM*, 1999, 173-275.

练习

- 10.1 我们希望提升例 10-1 中估计的存储器平均时延。假设设计存储器单元时使用 DDR SDRAM 来代替 SDRAM，请重新估计存储器平均时延。
- 10.2 计算例 10-1 中的预估平均时延适用于计算峰值性能。假设在最坏场景中，请求事务和响应事务均不可重叠。请重新计算预估的存储器平均时延，假设每次 SDRAM 访问（从行地址发射到第一个数据项出现在总线上）需要 5 个 SDRAM 时钟周期，忽略使一行无效所需要的时间。
- 10.3 考虑 CPU 循环 N 次访问下列 4 个存储器地址，假设存储器地址分割为标记、槽号和偏移，如图所示，完成以下内容：

0x3C1C (16 位地址)
0x0421
0x041F
0x0C88

| 标记 | 槽号 | 偏移 |
|----|----|----|
| 6 | 6 | 4 |

- a. 确定在第一次循环中缺失的次数，假设 cache 初始化为空。
- b. 确定循环 N 次总的缺失次数，假设 cache 初始化为空。
- c. 假设 cache 映射采用两路组相联。计算标记、组号、偏移域的大小，确定在第一次循环中缺失的次数，假设 cache 初始化为空。
- d. 确定循环 N 次总的缺失次数，假设 cache 初始化为空，使用循环替换策略。
- 10.4 使用以下地址重新完成 10.3 中的问题 a ~ d。
- 0x0C1C (16 位地址)
0x0521
0x041F
0x4D28
- 10.5 使用以下地址重新完成练习 10.3 中的问题 a ~ d。
- 0x3C1F (16 位地址)
0x042C
0x0460
0x3C1D
- 10.6 使用以下地址重新完成练习 10.3 中的问题 a ~ d，但是假定对于 c 和 d 部分采用 4 路组相联 cache。

0x3C17 (16位地址)
0x3817
0x3917
0x1C17

- 10.7 假设主存储器为 64KB, cache 大小为 4KB, 块大小为 16B, 对于直接映射 cache, 确定标记、槽号和偏移域的大小。
- 10.8 考虑一个 4 路组相联 cache, 讨论实现以下替代算法的复杂性 (如硬件需求)。
- a. 轮转方式: 循环选择一组中的每一行, 先进先出方式 (例如, 放置到槽 0、1、2、3、0、1、2 等)。
 - b. LRU: 替换最近最少使用槽。(提示: 考虑一个 4×4 矩阵、矩阵中每个点为一个记录, 记录大小为 1 位的 4 元素 LRU 算法, 假设 4 个槽标号为 0 ~ 3, 每当一个槽被访问时, 矩阵中相应的 1 行置为 1, 再将相应的 4 列置为 0。例如, 若第一次槽 0 被访问, 矩阵第 0 行将变为 $(0111)_2$, 矩阵其他几行保持为 $(0000)_2$ 。假设接下来访问槽 2, 按顺序, 矩阵中每行为 $(0101)_2$ 、 $(0000)_2$ 、 $(1101)_2$ 和 $(0000)_2$ 。LRU 算法将选中行值为 0 $(0000)_2$ 的槽。)
 - c. 随机替换: 在每个组中, 随机替换某一个槽。
- 10.9 考虑图 10-10 中的直写协议。请确定当高级语言程序语句 “A = 1;” 第一次执行时 FSD 状态转移。
- 10.10 考虑图 10-11 中的 MESI cache 协议。请确定当高级语言程序语句 “A = A + 1;” 第一次执行时 FSD 状态转移。
- 10.11 考虑在访问未共享的变量时, 发生的 MESI $I_{RM} \rightarrow E$ 和 $E_{WM} \rightarrow M$ 转换, 请陈述在编写多线程程序时, 如何使用以上信息来使程序在多核或多处理器系统中运行更加有效。
- 10.12 考虑一个包含 C0 和 C1 两个 MESI cache 的双处理器系统, 假设处理器执行两个线程 T0 和 T1, 其共享变量 A, 请概括 MESI 转换 $E_{XWH} \rightarrow I$ 发生时的场景, 假设存储器块 B_A 包含 A。
- 10.13 考虑图 10-9 中的系统和以下两个线程 T0 和 T1。假设最初 $x = 0$ 且 $y = 0$, P0 执行 T0, P1 执行 T1, B_x 包含 x, B_y 包含 y。使用以下表格说明 T0 和 T1 执行时在 cache 中的块 B_x 和 B_y 的状态转换。在汇编代码列表中, 与存储器相关执行的执行顺序如表所述。例如, T0 先执行 “STA (y)” (//1), 接着 T1 执行 “LDA (x)” (//2) 等等。并说明在每种情况下存储器更新或最终更新的次数。

| 线程 0 | | 线程 1 | |
|--|--|--|--|
| 程序代码 | 累加器 ISA 汇编代码 (第 8 章) | 程序代码 | 累加器 ISA 汇编代码 (第 8 章) |
| y = 1; x = 1; | LDA 0 STA (y) //1 LDA 1 STA (x) //3 | while (x == 0) { } //wait y = y + 1; | L1: LDA (x) //2, 4 CMP 0 JEQ L1 LDA (y) //5 ADD 1 STA (y) //6 |
| 线程: 指令 | 在 C0 中的块状态 | 在 C1 中的块状态 | 更新 cache 或存储器或两者都更新 (如果有) |
| T0: STA (y) //1 T1: LDA (x) //2 T0: STA (x) //3 T1: LDA (x) //4 T1: LDA (y) //5 T1: STA (y) //6 | | | |

- a. 直写协议
- b. 采用 MESI 协议
- c. 采用 MESIF 协议
- d. 采用 MOESI 协议

10.14 再次完成练习 10.13，但是本次与存储器相关指令执行顺序有变，顺序如下表所示：

| 线程 0 | 线程 1 |
|--------------|--------------------|
| 累加器 ISA 汇编代码 | 累加器 ISA 汇编代码 |
| LDA 0 | L1: LDA (x) //1, 4 |
| STA (y) //2 | CMP 0 |
| LDA 1 | JEQ L1 |
| STA (x) //3 | LDA (y) //5 |
| | ADD 1 |
| | STA (y) //6 |

- 10.15 简略解释当一个 cache 变为一个热点时，为何会增加平均存储器时延。
- 10.16 在下列结构中，陈述存储器何时进行更新：
- a. 采用 MESI 协议的基于总线的 UMA
 - b. 采用 MESIF 协议的 NUMA 结构
 - c. 采用 MOSEI 协议的 NUMA 结构
- 10.17 假设系统含有 16KB 的虚拟存储空间，页大小 16B，2KB 的物理存储器，完成以下内容：
- a. 确定虚拟页数量和物理页数量。
 - b. 假设页表中每条记录为 2B，页表最大大小为多少？
 - c. 请设计一个页表结构来转换 16 位的虚拟地址为 11 位的物理地址。
- 10.18 考虑一个 TLB，回答以下问题：
- a. 简单解释采用 TLB 的目的（如：当不采用 TLB 时的区别）。
 - b. 解释为何 TLB 需采用全相联的 cache（如：当采用直接映射的 cache 时有何不同）。
- 10.19 讨论在硬件中使用一个使用位（u）而不是 LRU 算法的好处，可参考练习 10.8。

计算机安全

- 10.20 计算机安全（虚拟存储器安全）：见练习 11.32（也可参见 11.11 节）。
- 10.21 计算机安全（对虚拟存储器的重放攻击）：见练习 11.33 关于如何检测对虚拟存储器的重放攻击（也可参见 11.11 节）。
- 10.22 计算机安全（存储器认证任务）：见练习 11.34（也可参见 11.9.2 节和 11.11 节）。
- 10.23 计算机安全（阻止信息泄露）：见练习 11.35 关于对存储器的随机加密（也可参见 11.11 节）。
- 10.24 计算机安全（程序安全执行）：见练习 11.37 关于针对安全执行如何建立可信程序。（也可参见 11.11 节）。
- 10.25 计算机安全（更有效阻止信息泄露）：见练习 11.38 关于如何使用更少存储器来阻止信息泄露（也可参见练习 10.22）。
- 10.26 计算机安全（支持安全执行模式的虚拟地址空间结构）：参见练习 11.39 关于如何分配多个虚拟地址空间（也可参见 11.11.8 节）。

计算机体系结构：安全

11.1 简介

纵观前文，我们的关注点主要为如何通过数字设计技术和计算机体系结构思想来提升计算机性能，因此，需要采用额外的技术和思想来设计安全计算机。如今，使用计算机的人和组织越来越多，这不仅仅产生了大量数据，很多新的应用软件也应运而生；但是某些软件极有可能存在安全漏洞，从而给从个人黑客到网络战争军队等一大批网络攻击者提供了机会和利益来源。许多组织，包括政府（如军事部门）、金融机构（如银行）、基础设施（如电网）、服务型行业（如法律部门）、商业企业（如电子商业）、工业企业（如工厂控制系统）和社交网络公司（如 Facebook）等在内都有需要保护的数字信息财产（程序、文件、数据等）。

资产，尤其是电力供应网络和工业控制系统，可能含有许多安全性问题 [1]。例如，其可以成为敌对国家网络战争的目标，敌对国家非常愿意花费时间和资源来发展一个复杂的网络攻击设备。所有的网络攻击问题可以被角色化为以下三种安全特性：

- **保密性**：资产具有隐蔽性，需阻止未被授权的访问（例如窃听），这种资产包括程序、文件和数据等，其存储在硬盘上，某些情况下，也包含存储在内存中的数据 and 指令。
- **完整性**：保证可检测到未授权访问对资产内容的修改，如果可以，阻止修改行为的发生。例如，代码注入将改变程序的完整性，对数据的非法修改将改变数据库的完整性。
- **可用性**：阻止对合法用户的服务进行延迟的攻击。这些攻击可能以多种形式存在，如使服务器负载超重。恶意攻击可以消耗如存储器和网络带宽等重要的资源，也可以通过使服务器负载超重等方式减慢或阻止计算机执行其预定服务。

三种安全属性的重要性是由组织资产的类型以及如何使用该资产所决定的。例如，对于银行家来说，银行账户的完整性要比保密性重要得多，保护资产的精确平衡要比资产的安全性重要。相似的，例如一个学生在大学校园中访问一台计算机，计算机的一个异常并因此造成的该计算机在几小时内不可用并不是非常重要。另一方面，对于政府机构，如军队，这三种安全属性都是必不可少的。

由于计算机安全涉及的范围太大，其包含的主题范围太广，所以在本章中，我们主要介绍与计算机结构相关的计算机安全话题。此外，即使是在与计算机结构相关的计算机安全范围内，该范围及其演化还是过于广泛，因此，我们在本章中主要介绍一些新的概念和正在研究的方法，具体介绍了与计算机结构相关的计算机安全概念并说明了一些新的具有启发性的解决方法，此外，包括信息流追踪法等研究方法请参考延伸阅读或其他书籍。信息流追踪法要求存储器中每一个危险的位或字都需要标记为安全的或不安全的。例如，通过 I/O 端口读入的数据将被标记为不安全的，而系统数据将被标记为安全的。当被标记的数据项进入 CPU 时，其将被追踪，并采用一些控制方式来阻止未授权的修改 CPU 状态（例如寄存器）的行为。这些控制方式需要额外的存储器空间来存储标记并需要对硬件设计（逻辑电路、数据通路和存储器组织结构）进行修改。

当对部分软件或硬件进行开发或安装时，可能导致一些安全问题。而如今，大多数软件公司和硬件公司都依赖于使用第三方模块来进行开发，而这些模块可能未正确设计或使用，很有可能包含一些木马病毒（非法代码或 HDL 模块），计算机安全问题更显得尤为重要。软件安全性策略和安全性机制通常是基于成熟的模型，例如在军队中使用的软件模块。而硬件安全性策略和安全性机制是基于一系列阻止攻击的技术。本章中将介绍一些已知的安全的软件模块的示例和采用该模块的程序，并介绍适用于硬件的安全策略机制。

虽然对于攻击者而言有无穷种方式来利用软件安全漏洞进行攻击，但是软件攻击通常是在存储器中插入无效数据或将数据从存储器中的一个位置复制到另一个位置。例如，思考一个简单的 C 语言程序，其调用“strcpy”库函数来复制其命令行内容到一个在子程序中本地声明的数组（缓冲区）中。在这种情况下，攻击者可以使用特定的参数值来“欺骗”CPU 并造成缓冲区溢出攻击 [2]。总体而言，攻击者可以使用在程序中静态分配的或动态分配的缓冲区来修改存储子程序返回地址等的存储器堆栈区，通过在缓冲区中嵌入一段恶意代码来改变在堆栈区的子程序返回地址等，使得系统运行这一恶意代码，允许工具或程序在系统特权模式可被使用。例如，使用软件攻击故意破坏系统或使系统超负荷，使得系统不可用或相对于正常状况处理事务效率过低。攻击者也可以获取机密文件、修改数据库、触发一个使硬件故障的硬件特洛伊病毒或泄露机密信息等。

464

当攻击者独占系统时，其也可以使用欺骗或其他技术来执行物理攻击 [3-5]。攻击者可以使用复杂设备来欺骗系统或观察信号，从而获取访问端口化设备的途径或在该设备上执行逆向工程。本章也将介绍包括欺骗技术在内的用于执行软件/硬件攻击的相关技术。

总体而言，因为一个系统有太多的安全性漏洞需要填补，所以不可能对一个安全系统中的每一个硬件、固件、软件都采用安全性设计、开发、安装，关键是采用可信计算基（TCB）来构建必要模块从而实现一个安全系统 [6]。TCB 指的是一组最小的硬件和固件以及其安全实现（设计、开发和安装）的要求，它的设计必须是安全和可靠的（即保持可信）。此外，根据系统对安全性的要求，TCB 不仅要包含实现安全性的软件，还要包含执行安全性策略的软件。下面是一个安全程序应用区域列表：

- 对于手持设备，与主计算机安全地交换数据并防卫物理攻击。
- 对于系统设计者，需实现安全策略机制来阻止对系统资源的未授权的访问，系统资源包括密码文件、系统堆栈存储区等。
- 对于系统设计者，需实现安全策略机制来阻止现有操作系统经常性受到损害 [3, 7-9]。
- 对于软件公司，可开发并建立独立安全的应用程序安全策略机制。
- 对于软件公司，安全地传送远程安装程序。
- 对于用户，可以文件、数据、图片等形式隐藏信息，并将其安全地存储在一个本地或远程磁盘驱动器上，安全地发送和接收邮件，执行安全的远程登录等。
- 对于公司，实现安全策略机制来阻止对其至关重要的商业资源的未授权的访问。这些资源包括个人数据、客户数据、知识产权等。
- 对于娱乐公司，只向已授权的手持设备发送付费产品及其相关信息。
- 对于云计算公司，向其客户提供有保证的计算服务。

465

最后，本章将介绍用于实现软件/硬件安全策略机制机密性和完整性的相关技术，并提供一个基于处理器和协处理器的 TCB 架构示例和其应用区域。

11.1.1 安全工程方法

图 11-1 展示了安全工程方法 (SEM)，为设计者提供了逐步的处理过程来确定潜在的威胁，开发所需的安全策略和机制，并设计、验证和评估计算机架构的安全性。对用户场景和潜在的安全问题的分析确定了安全风险的范围。用户场景通常由应用程序决定，可能涉及多种系统，比如嵌入式系统、实时系统和分布式系统，也可能覆盖很多行业，包括 IT、制造业、医疗保健、商业等 [10-11]。对威胁模型、安全策略和安全机制的分析可确定一系列可能的威胁和对每种威胁的安全策略和处理机制。

为了更好地理解 SEM，表 11-1 使用大学资产中的学生成绩为示例说明了基于学生成绩的安全策略机制开发过程。

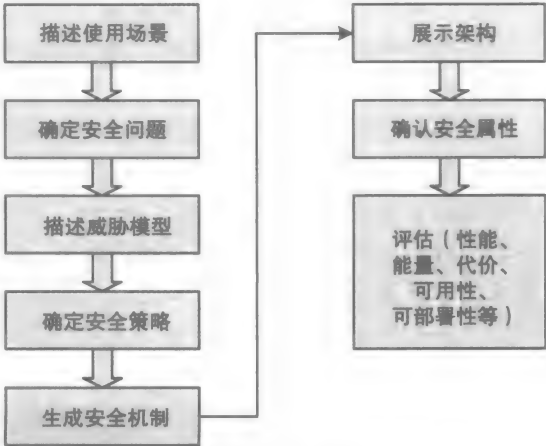


图 11-1 安全工程方法 [12]

466

表 11-1 开发一个基于学生成绩的安全机制来改变学生成绩

| 方 法 | 描 述 |
|------|---|
| 使用场景 | 如何改变学生成绩 |
| 安全问题 | 对学生成绩未授权的修改行为 |
| 威胁模型 | 错误的成绩 |
| 安全策略 | 只有教授该课程的授课老师可修改学生成绩 |
| 安全机制 | 授课老师为其学生分配一个新的成绩 部门主席确认成绩的修改 一个全职的员工（不是临时工）在安全的大学数据库系统中录入成绩 |
| 架构 | 生成一个成绩变化表，为学生姓名、学生 ID 号、课程号、学期、改变成绩的原因、指导老师和部门主席的签名分配空间。此外，使用一个完整的方式来传送邮件，传送方式包括使用校园邮件系统或手送到学生记录办公室 |
| 验证 | 机密性：成绩变化表只可被科员或职工手动修改，并对其他学生保密 完整性：主席的签名授权了一个完整的成绩变化表 可用性：依赖于教授改变学生成绩的决定 |
| 评估 | 成绩改变表的代价，需要完成和处理成绩变化表的时间，等 |

在表 11-1 中，用户场景可以表示为一系列的用例，威胁模型可表示为一系列的威胁向量。总而言之，一个威胁向量可被视为一个引导对个人的、商业、IT 或其他领域信息的偷取、毁坏、使失效的攻击路径。

表 11-2 列出了数据存储和远程服务器连接的两个计算机用例。病毒和其他类型的恶意软件，以及被盗窃或受到其他损害的计算机，具有一定的安全风险。一个恶意软件可以删除、修改或偷取存储在硬盘上的数据。一个包含有价值信息的计算机（如台式机、笔记本等）或手持设备（如智能手机）的丢失或窃取也具有一定的安全风险；存储在硬盘或 flash 存储器中的数据也是易被篡改的。此外，如果丢失的设备属于公司，这给攻击者提供了访问公司服务器的不正当的途径，可能对公司资源造成损坏、对公司重要文件进行删除或修改或窃取

467

公司商业专利等。

表 11-2 用于两个计算机使用场景的五步 SEM 过程

| 使用场景 | 安全问题（漏洞） | 威胁模型 | 安全策略 | 安全机制 |
|--------------|---|------------|------------------|---|
| 数据存储 在硬盘上 | 文件存储、用户认证数据（如用户名 / 密码）等，这些数据存储在硬盘上，易被修改 | 数据泄露 | 保持数据机密性 | 保持硬盘上的数据处于“锁住”状态，需要“钥匙”来解锁，没有钥匙，数据保持机密（以不可知的方式存储），钥匙不存储在硬盘上，而存储在系统的硬件中 |
| 网络身份 认证 | 存储在硬盘上的网络认证数据易被修改 | 对服务器未授权的访问 | 认证平台（如笔记本），而不是用户 | 对公司的每个笔记本电脑使用双端口锁，使用一把“钥匙”来“锁”住存储了网络认证数据的磁盘，并隐藏钥匙使之与计算机的硬件相关，维护另一把与公司或与公司信任的公司相关的“钥匙”，使用第二把“钥匙”来解锁收到但锁住的网络认证数据，当笔记本丢失或被偷窃时使第二把钥匙为无效 |

* 现在安全机制由通俗术语陈述

表中两个用例都包含了潜在的安全漏洞、一个威胁模块、一个安全策略和一系列安全机制。然而，安全机制使用俗语来描述，更精确的解决方式将在之后给出。

11.1.2 威胁类型

对安全的威胁可被分为两大类：可操作的威胁和可发展的威胁。

1. 可操作的威胁

可操作的威胁与资产类型和使用场景相关，如表 11-2 所示。可操作威胁也与用于实现安全机制的工具相关。当安全机制设计得很好时，其采用的工具（如“锁”）可能较差；例如，“锁”可能是低质量的或容易解开的。关于可操作威胁的其他示例将在后文讨论。

设备，如由公共事业公司安装在家庭中的智能电表或是在紧急情况下由军队人员或消防员或紧急医疗人员使用的便携式设备，可能面临额外的威胁。这些设备需要使用安全的通信通道来与主计算机通信，但攻击在某些情况下，如在紧急情况下，可能会导致通信干扰。

468

物理上可访问的远程安装设备和便携式设备也是物理攻击的目标。例如，安装在汽车中用于读取汽车行驶里程数的嵌入式设备可能被攻击从而篡改某些旧车辆的行驶里程数。相似的，高科技的便携式设备也可能被黑客攻击从而修改或逆向工程的功能。

2. 可发展的威胁

这些威胁依赖于用于软件和硬件开发的可信模型，以及交付和安装软件。一般情况下，不完整的规范、不正确的实现和不正确的安全策略和机制是软件、硬件和固件漏洞的三个来源。通常，这些漏洞是无意产生的，但有时是设计师故意设计的。

例如，大型集成芯片（集成电路）的设计，特别是处理器，可以包括设计师注入的恶意电路，如在设计 [13] 中为了某种目的添加额外的硬件描述语言（HDL）代码。总体而言，故意造成的漏洞在验证过程中更难被检测。多种因素，包括在硬件设计中对第三方“软件”组件（如 Verilog 模块）越来越多的使用增加了可被攻击的漏洞。

11.1.3 访问控制和类型

表 11-2 中描述的数据存储安全机制是设计用来保护存储在硬盘上的数据避免其被恶意

软件攻击和物理攻击。其他的使用场景要求机制限制用户访问。例如，在一个组织中，需指明哪些人员可以访问雇员的个人信息（例如薪水、社会安全号码等），可能只有雇员的管理者可以被允许检验雇员的个人信息，雇员信息对其他雇员必须保密。

相似的，在一个计算机系统中，只有系统程序，而不是应用程序，可以访问系统数据。这样的安全机制被称为访问控制。使用访问控制列表（ACL），例如 Linux/UNIX 系统中使用 ACL，决定了每个用户可访问的文件和文件夹。用户可使用命令“chmod”来给各个文件和文件夹分配读（r）、写（w）和执行（x）等权限。例如，对于文件 foo，其最开始分配所有用户的读写（rw-）权限，使用命令“chmod 640 foo”给拥有者（如 Joe Smith）分配读写（通过二进制中的 110 表明 rw-）权限，给该组中的其他用户分配只读（通过二进制中的 100 表明 r--）权限，给其他用户分配不可访问（二进制中 000 表示 ---）。如下所示：

```
>ls -l
-rw-rw-rw- 1 smithjoe faccsc 5 May 22 12:32 foo
>chmod 640 foo
>ls -l
-rw-r----- 1 smithjoe faccsc 5 May 22 12:33 foo
```

469

通过使用 ACL，即使文件和文件夹不是隐藏的，其仍然是秘密的并且不可被其他用户访问。Linux/UNIX 的 ACL 是一种自主控制访问的示例，因为其每个用户都可决定给其文件和文件夹分配怎样的访问权限，另一方面，基于规则的强制访问控制对一个组织中的所有资产（对象）和主体（人和程序）强制执行一组保密性和完整性的安全规则。例如，当只有雇员的主管可被允许检查用户的个人信息，这种强制访问控制被称为权限表（CL）[14, 15]。每个主体被分配一系列的权限——组织中所有对象允许执行的一系列动作。另一方面，由对象来组织，而不是由主体来组织的强制访问控制被称为强制 ACL，与早先讨论的 Linux/UNIX 中任意 ACL 示例相似。为了阐述示例，表 11-3 说明了一个由大学成绩政策生成的访问控制矩阵。

表 11-3 学生成绩的访问控制矩阵

| 主体：人 | 对象：成绩 | | |
|--------|-------|------|------|
| | 课程 A | 课程 B | 课程 C |
| 主席：p | R | R | R |
| 教授：x | R, W | | R, W |
| 教授：y | | R, W | |
| 职工雇员：e | R | R | R |
| 学生：s1 | r | | r |
| 学生：s2 | | r | r |
| 学生：... | ... | ... | ... |
| ... | ... | ... | ... |

在该表中，每一行是一个主体（教授、学生等），每一列是一个对象（课程），矩阵的主体是对学生成绩的一系列访问权限。访问权限被定义为读（R 或 r）、写（W 或 w）、可读可写、既不可读也不可写。大写字母 R 和 W 表明对某一个单独课程的所有成绩的读权限和写权限。小写字母 r 和 w 表明对某一课程的分别对每一个成绩的读或写权限。在表中，既不可读也不可写权限用空白表示。该表中，包含一位主席 p、两位教授 x、y、两位学生 s1 和 s2、一位职工雇员 e 和 3 种课程分别标记为 A、B、C。从表中可看出，教授 x 可为选修课程 A 和

课程 C 的学生分配 (R/W) 成绩, 教授 y 可为选修课程 B 的学生分配 (R/W) 成绩, 学生 $s1$ 可以读取其课程 A 和课程 C 的成绩, 学生 $s2$ 可以读取其课程 B 和课程 C 的成绩。该矩阵中没有单独的写权限。

470

如果一个访问控制矩阵是根据其行内容来存储, 该矩阵将生成一个权限表。例如, 主席 p 、教授 x 和学生 $s1$, 每个人都含有下列的权限表, 关系如下所示:

```
p: {(A, R), (B, R), (C, R)}
x: {(A, R), (A, W), (C, R), (C, W)}
s1: {(A, r), (C, r)}
```

例如, 给教授 x 分配的权限是对课程 A 和课程 C 的所有成绩的读权限和写权限。而给学生 $s1$ 分配的权限是对其选修的课程 A 和课程 C 的读权限。

另一方面, 如下对课程 A 和课程 B 的权限分配, 当访问控制矩阵是根据其列内容来存储的, 该矩阵将生成一个强制 ACL。如下:

```
A: {(p, R), (x, R), (x, W), (e, R), (s1, r), ...}
B: {(p, R), (y, R), (y, W), (e, R), (s2, r), ...}
```

例如, 课程 A 的访问权限表表明主席 p 和职工 e 对选修课程的所有学生的成绩具有读权限, 教授 x 对选修课程的所有成绩具有读权限和写权限, 学生 $s1$ 只对其个人的成绩具有读权限。因为一个 ACL 是采用面向对象机制的, 其更容易改变一个对象的权限——例如, 将学生 $s3$ 加入课程 A 中。

当一个程序希望访问一个对象 (如一个文件、数据项、确定的存储器地址、网络连接、USB 端口等) 时, 其必须被包含在该对象的访问列表中, 否则, 访问将被阻止。相反, CL 是面向主体的, 故而是一个主体可以传送其分配的全部或部分权限表给另一个主体。例如, 教授 x 可以传送其权限数据项 (C, W) 给主席 p , 从而使得主席 p (而不是教授 x) 具有给选修课程 C 的所有学生分配成绩的权限。

虽然基于 ACL 的系统容易实现, 但基于 CL 的系统可以提供更好的保护, 用户或进程只可以访问其权限列表中的对象。基于 CL 的系统也可以提供更精细的保护, 传送的权限可被限制为一部分数据、存储地址、任务等。例如, 教授 x 可以只传送分配单个成绩的权限, 如 (C, w(i)), 给主席 p , 其中索引 i 是用于确定一个特定的学生, 例如选修课程 C 的学生 $s2$ 。然而, 一旦一个权限被传送, 无法控制该权限再次被传送给另外的主体。一个系统可以使用混合的访问控制方式来获取 ACL 和 CL 模式的优点, 关于其他访问控制方式, 如基于角色的访问控制和基于发起人的访问控制, 将在下文阐述。

11.1.4 安全策略模型

具有保密性和完整性的安全策略模型必须能够为一个计算机系统的所有硬件和软件组件创建一个完整的安全外围环境。例如, 考虑火焰病毒, 其可以启动一个计算机的音频系统从而通过网络来窃听并传输办公室等机密的谈话内容、捕获屏幕画面、记录键盘按键, 甚至偷取计算机附近开启蓝牙功能的手机中的数据。相似的, 如震网病毒等网络武器可通过通用串行总线 (USB) 接口进入工业控制系统中, 并改变控制系统的操作规范, 例如, 其可以使一个工业发动机运转过快从而造成设备损坏。

471

强制访问控制一般是基于一些已被验证机密性和完整性的安全策略模型, 例如使用于军队和商业环境中的安全策略模块。安全策略模块具有**多级别** (分等级的) 或**多方面** (区划的)

的特点。下面是对一些著名的多级别和多方面的安全策略模型的描述。

1. 多等级模型

多等级模型主要用于采用自然等级访问信息的场景，如军队或医疗办公。在军队中，主体（如人）和对象（如文件）都进行了分类，如按“最高机密”、“机密”、“保密”和“未分类”分类。在医疗办公中，只有医生可被允许访问特定病人的医疗记录。通过安全策略模型可控制拥有对每种文件或医疗记录读写权限的人员。

Bell-LaPadula (BLP) [16] 是一种用于军队中为了提高机密性的多等级安全模型。BLP 中的“不可向下写”策略会阻止含有更高许可的职员写或添加一个较低分类等级的文件。此外，该策略阻止了信息流从高等级对象向低等级对象的转移。例如，一个拥有最高机密权限的腐败的军队将军对一个已分类文件的读操作和传输信息到一个未分类文件的操作将被阻止。这被称为 BLP 模型的 *- 属性。在一个计算机系统中，*- 属性可阻止将军复制一个已分类文件到一个 USB flash 存储器中，这个存储器在该系统中可能具有较低等级。（此外，所有的高等级人员可能不允许携带智能手机到他们的办公室中。）

BLP 的“不可向上读”策略将阻止低等级的具有较低许可的主体访问具有较高等级的对象。该策略也会阻止一个军队中未被分类的职员对最高机密的文件的读取行为。此外，该策略将阻止从网络上下载恶意软件，该软件在系统中拥有较低的许可，对高等级对象（如密码文件或机密的用户文件）的访问行为。

完整的 Biba 模型 [17] 采用“不可向上写”和“不可向下读”策略。主体和对象都分配了完整性级别（或标记），例如，系统文件被标记为高等级而网络文件被标记为低等级。“不可向上写”策略可以阻止从网络上下载的低等级的恶意软件对高等级系统数据的修改，例如修改一个递归子程序存储在系统堆栈上的返回地址。“不可向下读”策略可以保证某程序一旦从网络上接收数据，立刻降低该程序的完整性级别。在这种情况下，即使恶意软件以某种方法获取到管理员权限（如调用一个 root 脚本），该程序的完整性级别依旧将被降低从而使其无法修改密码文件，但是，在这种情况下，即使 BLP 策略仍在运行，恶意软件依旧可以读该密码文件，并通过网络连接传输该密码文件。

当 BLP 策略和 Biba 策略都被使用，“向上写”“向下写”“向上读”“向下读”都是不被允许的，因而，这两种策略的混合机制产生了一种更为强大的安全模型。然而，这两种策略的混合可能潜在地对某些应用造成访问限制，例如一个数据必须共享的数据库。LOMAC 是一种采用 Biba 策略的实例，其是一种对商业的 Linux OS 的强制访问控制 [18]。

2. 多边模型

多边模型主要用于访问的信息不分等级但相互隔离的情况，例如，在商业交易或产生利益冲突等活动中的职责分离。

BLP 模型的保密性特征并不适用于不分等级的服务型企业，如律师事务所、会计师事务所、广告公司等含有竞争客户的企业。这些服务型行业的职员可能会收到一些敏感的客户信息，而这些信息是必须被保护的并且不可以与同行业的其他顾客分享。这种类型的信息保密性并不是多等级的，而是多边的。中国防火墙 [19] 的多边模式是设计用来阻止利益冲突。

例如，一个有来自不同行业的客户（如银行、石油公司等）的法律公司，不应允许其雇员从事有利益冲突并可能导致客户的商业信息（如花旗银行）泄露给客户同行业（例如，威尔斯法戈银行）的其他人员的活动。

相似的，Biba 这一多等级完整性模型，不可工作在商业环境中。例如，输入购买商品

命令的主体（公司的雇员或软件）应该与接收商品和为商品支付的主体不同。在这种情况下，使用一个多边的完整模型，如 Clark Wilson [20] 模型，实现了在商业交易中的职责分离的原则。表 11-4 是对这些安全模型的总结。

表 11-4 多级安全策略模型和多边安全策略模型

| 安全策略模型 | 描 述 | | 示 例 |
|--|----------------------|--|--|
| 多级 | BLP: 保密性 | “不可向上读”: 当且仅当 $CL(S) \geq CL(O)$ 时, 主体 S 可以读取对象 O 内容 | 一个 LC-S (如下载的软件) 不可访问一个 HC-O (如系统数据) |
| | | “不可向下写”: 当且仅当 $CL(S) \leq CL(O)$ 时, 主体 S 可写对象 O | 一个 HC-S (如系统程序) 不可以通过 LC-O (例如网络连接) 传输数据 |
| | Biba: 完整性 | “不可向下读”: 当且仅当 $IL(S) \leq IL(O)$ 时, 主体 S 可以读取对象 O | 一个 HC-S (如系统程序) 不可以访问 LC-O (如通过 USB 设备或从网络上下载的数据) |
| | | “不可向上写”: 当且仅当 $LI(S) \geq IL(O)$ 时, 主体 S 可以对对象 O 执行写操作 | 一个 LC-S (如下载的软件) 不可以修改一个 HC-O (如系统存储器堆栈) |
| 多边 | 中国防火墙: 保密性 | 防止利益冲突 (COI); 既需要自主访问控制也需要强制访问控制; 将对象 (O_s) 组织为 COI 的集合; 可以选择访问类型 (如自主访问控制) 并访问该 COI 集中的一个对象; 还引入了时间参数 | 访问对象的 $O_i \in COI_k$ 的主体 S , 在一段时间内无法再次访问对象 $O_j \in COI_k$ 。两个在同一个法律公司工作的离婚律师, 每一个代理某对已婚夫妇的一员, 两者不能够访问其他成员的离婚文件 |
| | Clark-Wilson 模型: 完整性 | 数据和事务的完整性; 执行职责分离; 如果注册交易 CT , 允许访问注册数据 CD_j , 将生成一个注册关系, 为 (CT, CD_j) | 如果主体 S_1 在对象 O 上运行 (例如, 订单) 并生成了有效的关系 $(S_1, CT = \text{“订单”}, CD = O)$, 然后 $(S_1, CT = \text{“接收”}, CD = O)$ 是一个无效的关系; 关系 $(S_2, CT = \text{“接收”}, CD = O)$, 其中 $S_2 \neq S_1$ 将依旧为一个有效关系; 因此, S_1 和 S_2 有各自的职责 |
| S: 主体 (人或程序); O: 对象 (文件、网络连接、USB 端口、数据等) | | | |
| LC: 低等级或低类别; HC: 高等级或高类别 | | | |
| CL: 保密性等级; IL 完整性等级; CT: 注册事务; CD: 注册数据 | | | |

11.1.5 攻击类型

可发展的威胁，如前面所讨论的，可以产生无意的或有时有意的以后门形式存在的漏洞。硬件的后门攻击通常是由于用于建立一个系统的硬件模块中存在一个或多个恶意电路（硬件木马）。鉴于现代集成电路的大尺寸，设计中的恶意电路是不太可能在验证过程中被发现的。此外，几乎所有现在制造的 FPGA 和其他一些芯片可能包含一个远程激活的“杀死开关” [21]。因此，硬件后门攻击可能会出现严重的安全隐患。当 IC 在一个系统中制造和安装之后，一个恶意电路可通过使用恶意软件或具有对系统的完全访问并执行一个触发程序来远程触发。

另一方面，如病毒和间谍软件等恶意软件的示例通常是利用软件中无意留下的后门来进行攻击。当专门的设备接入硬件中时，物理攻击可能在其正常运行中改变其行为。软件攻击和硬件攻击使用的攻击机制很相似。

除了以上攻击类型之外，还有诸如侧信道攻击等其他类型的攻击方式，其并不是通过后

473
}
474

门的方式而是通过收集程序运行时的信道信息的方式来进行攻击。程序执行时间是一种侧信道攻击收集的信息示例，这种攻击被称为**定时攻击**（参见 11.5.3 节），在程序执行过程之中将自然释放电磁辐射和声学信号 [22, 23]，此外还有基于 cache 的侧信道攻击 [24]。

11.2 硬件后门攻击

图 11-2 显示了硬件木马程序的三种触发机制。在这种情况下，攻击者通过一个触发输入使得多路复用器（MUX）选择恶意电路而不是原来的电路产生的结果。触发输入可能是数据、控制（例如，指令），或与时间相关。此外，它可以由一个数据项或一个控制项或两者都有或一个数据序列或控制输入或两者都有或一个计数器（定时器）来触发攻击。后者的情况被称为一个**定时炸弹**。

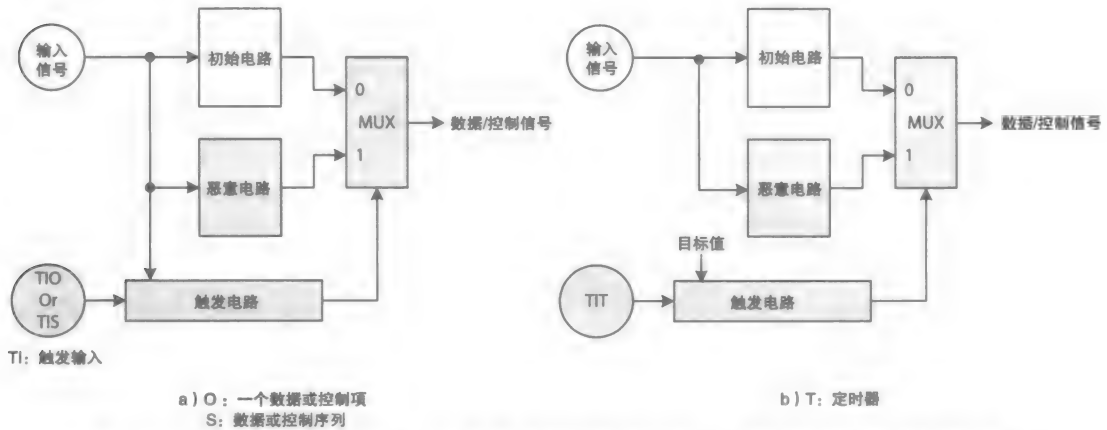


图 11-2 硬件木马示例 [12]：a) 数据或控制触发的木马；数据或控制序列触发的木马；b) 定时器触发的木马（定时炸弹）

此外，攻击可分为不可计算的和可计算的，**不可计算的**的攻击的主要目标包括存储器、寄存器、MUX 和其他不对数据进行计算而仅仅存储或路由数据的设备；**可计算的**的攻击的目标主要为算术逻辑单元（ALU）、译码器、有限状态机（FSM）等对输入数据进行处理的设备。

此外，硬件后门攻击可能会采取多种形式。例如，如果图 11-2 中的多路复用器的输出是一个数据项并且攻击将导致 MUX 的输出值发生变化攻击，这种攻击被称为一个**腐蚀者攻击**。另一方面，如果 MUX 输出位代表控制信号并且可能导致更多的事件发生，这种攻击被称为**发射者攻击** [13]。

一个后门攻击可能会改变和简化通过硬件实现的加密算法，产生更多的高速 cache 流量，造成计算错误，消耗更多的能源等。下面的部分提供数据、控制和定时器后门攻击的例子。

11.2.1 数据和控制攻击

图 11-3 说明了两个硬件木马的例子，一个采用单指令触发机制，另一个采用三指令序列的触发机制。在图 11-3a 中，木马是由一个使用一个特定操作数“11001100...”的 ADD 指令触发；在图 11-3b 中，木马由三条指令序列“ADD 0”、“ADD 0”和“ST 0”触发。以上指令仅仅为一些在累加器 ISA（第 8 章）类型中使用的示例。这两种类型的木马通常会使用在测试过程中不太可能检测出来的触发输入来选择。例如，在电路测试过程中随机选择一

条使用操作数为“11001100...”的“ADD”指令或选择三条指令“ADD 0”、“ADD 0”和“STM 0”的概率有多少？

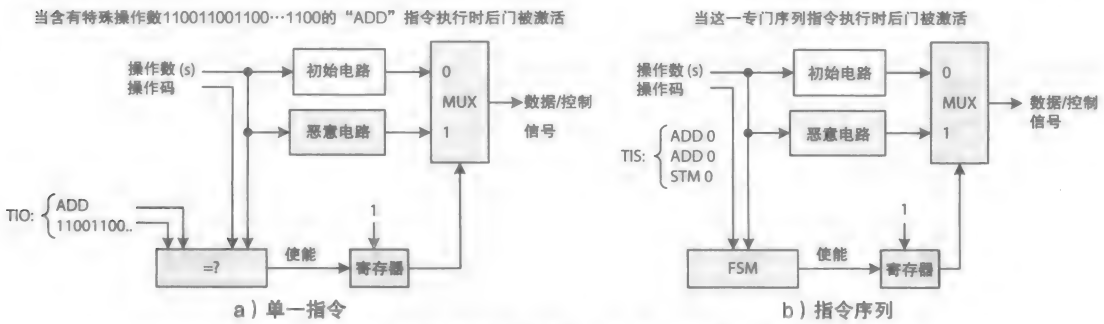


图 11-3 硬件木马触发机制示例：a) 单一指令；b) 指令序列

两种硬件木马都需要攻击者访问硬件，通过直接连接或恶意软件，来输入必要的触发输入。对于图 11-3a 中的电路攻击者必须可执行“ADD”指令而对于图 11-3b 中的电路攻击者必须可执行三条指令序列来触发攻击。

11.2.2 定时器攻击

如图 11-4 所示，定时器触发木马并不需要攻击者访问硬件。一旦计数器的计数降为 0 时，木马将被激活。因为在大多数测试用例中，尤其对于那些随机的、测试不长的并且要求几百万周期的测试用例，只要将计数器的值设置为足够大就可以在测试过程中避免被检测出来。

在图 11-4a 中，攻击者将初始 ALU 生成的数据转换为有恶意 ALU 输出的数据，从而破坏输出结果。在图 11-4b 中，当程序向特定的存储器地址写数据时，攻击者可以改变 cache 控制器的信号，导致在下层存储器中进行其他动作，从而造成信息泄露。其他类似的木马进行攻击的用例留给读者想象。

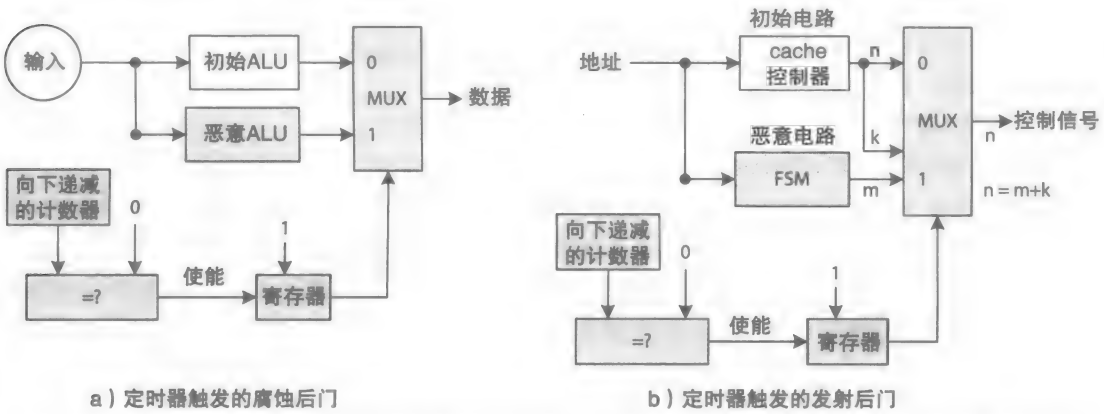


图 11-4 基于定时器的腐蚀后门和发射后门示例：a) 定时炸弹腐蚀后门，b) 定时炸弹发射后门

11.2.3 安全策略机制

在电路设计周期中检测硬件后门的方法之一是利用这样一个事实：硬件设计师团队通常是分层组织，对于一个复杂的 IC（如处理器），每个团队都不可以设计该芯片中的所有必

要模块。每个团队都可以设计自己的模块，也可能需要将他们的模块与其他内部或第三方的“软件”模块互连，如一些硬件描述语言（HDL）模块。然而，虽然一些内部模块可能必须通过代码审查和其他质量控制技术以确保它们的可信性，但是其他模块仍然可能包含木马，导致安全问题。

下面的章节给出了硬件后门安全策略机制的示例。

1. 数据混淆

防止一个单一输入触发攻击的安全策略是数据保密性。然而，根据不可信的电路模块的类型，需要不同的策略机制。如果不可信的模块是不可计算的，如图 11-5 所示，只有一个简单的数据保密技术被称为数据混淆 [25] 是用来防止攻击的。在图中，从可信模块中输出的 $(00110011)_2$ 在输入到不可信模块之前将与一个随机数字 XOR，从而混淆该数据。在这种情况下，当攻击者在不可信不可计算模块中生成触发输入 $(00110011)_2$ 来激活模块中的木马时，实际生成的触发电路的数据变为 $(10010001)_2$ ，从而阻止了本次攻击。

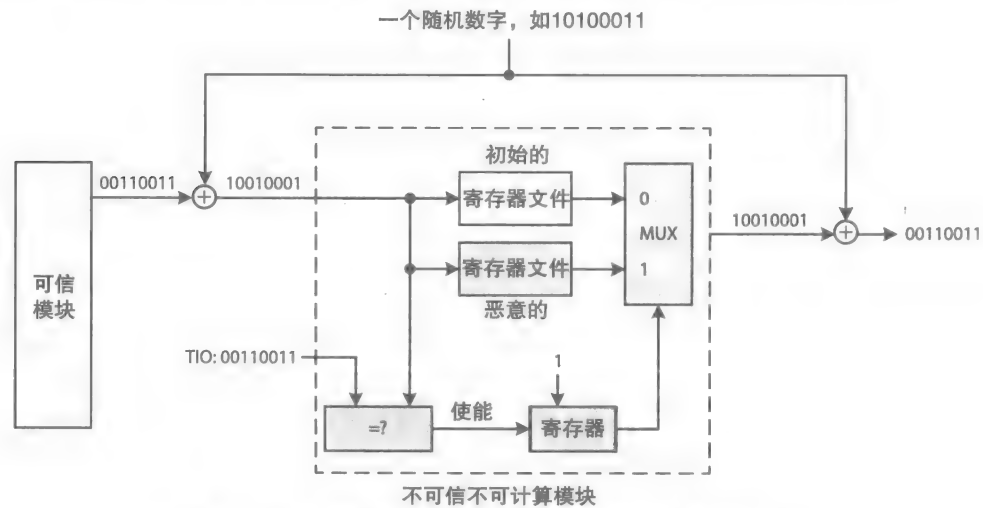


图 11-5 使用数据混淆技术的不可信可计算模块接口

2. 同态加密

然而，当在可计算模块中采用单个输入来触发木马时使用数据混淆技术有一点困难。例如，如果不可信可计算模块执行平方函数，一种被称为同态计算，也被称为同态加密的复杂数据混淆技术将被使用 [25-26]。当公式 (11-1) 成立时，两个函数 f 和 g 被称为同态关系。

$$f(g(x), g(y)) = g(f(x, y)) \tag{11-1}$$

例如，如果 f 是乘法函数， g 是平方函数，则 f 到 g 的同态关系如下所示：

$$\begin{aligned} f(x, y) &= xy \\ g(x) &= x^2 \\ g(y) &= y^2 \end{aligned} \tag{11-2}$$

所以，

$$f(g(x), g(y)) = f(x^2, y^2) = x^2 y^2$$

等价于

$$g(f(x, y)) = g(xy) = (xy)^2 = x^2 y^2$$

当 $x = 3, y = 2$ 时,

$$3^2 * 2^2 = (3 * 2)^2$$
$$9 * 4 = 6 * 6 = 36$$

假设一个设计团队在处理器的设计过程中在不可信 HDL 模块中加入浮点平方函数。在这种情况下, 为了防止单一输入触发攻击, 设计团队需要在硬件设计中包含之前所述的同态加密, 或者在处理器系统中使用软件方式实现同态加密, 如下:

```
float homomorphic_encrypted_square(float x)
{
    float y;
    y = random();
    return (square(x * y) / (y * y));
}
```

477
2
478

在这种情况下, 因为 y 是随机生成的并且是用于掩盖 x , $x * y$ 的值也是随机的, 使得其不太可能触发输入。平方函数计算的是 $(x * y)^2$, 而不是可能触发攻击的 x^2 。而值 x^2 之后通过计算 $(x * y)^2$ 除以 $y * y$ 安全得到。

同态加密的优点在于其对计算的操作是基于加密(隐藏)的数据而不是最初已知数据。例如, 为了说明另一个应用程序以及同态计算的优点, 假定在之前的代码中的平方函数代表存在同态关系的函数调用, 假设函数只在一个远程计算机上可用(如: 云), 并且用户并不希望通过网络传输数据 x (表示某些机密输入, 例如, 医学数据), 也不希望远程计算机访问数据 x 。在这种情况下, 通过在远程函数中对输入数据的隐藏(加密), 用户可以在不向外界暴露真实的输入数据并且可以安全地使用远程函数。关于同态计算的更多潜在问题请参考练习部分。

此外, 对于某些函数如平方根, 其在 x 和 y 都为非负数时保持同态关系 $\sqrt{xy} = \sqrt{x}\sqrt{y}$, 但当 x 和 y 为负数时不保持同态关系, 例如 $\sqrt{(-1)(-1)} \neq \sqrt{(-1)}\sqrt{(-1)}$ 。

在理论上, 就电路规模而言, 采用硬件同态加密/解密算法的可信模块实现成本过大而不可接受。

3. 序列器

如果是通过一个特定的输入序列来触发攻击, 防止这种攻击的安全策略是改变其输入给不可信模块的输入数据的顺序。对输入数据重新随机排序或在正常输入中插入虚拟输入可用于防止输入序列攻击。许多现代的处理器的, 如采用动态调度的超标量处理器(第8章), 已经通过对指令进行重新排序等方式来提高性能; 因此, 它对这项任务可能带来一定的随机性。然而, 对输入数据的随机重排可能对存储读操作不起作用。在这种情况下, 因为数据依赖而使得对输入数据随机重排不起作用时, 需要使用虚拟输入。例如, 为了改变存储器访问的顺序, 通过插入包含伪随机生成的存储器地址的 load 指令到一大序列 load 和 store 指令中从而阻止攻击被触发 [25]。

4. 电源复位

另一种定时炸弹攻击的安全策略是阻止计数器到达触发值。完成这种工作的一种策略机制是频繁地对不可信模块进行电源复位。对不可信模块进行电源复位的频率是由测试时模块正常工作的时钟数决定的。由于存在不可信模块中的后门类型不可知, 数据混淆、输入序列重排、电源复位等都必须使用从而达到阻止攻击的目的。

479

5. 副本

当攻击阻止技术无法工作时, 一种额外的但代价昂贵的解决方式是使用副本。图 11-6

说明了一个使用模块副本技术的示例，其中两个模块 X 分别由两个团队 A 和 B 设计。将对两个副本的输出进行比较。若输出结果相同，认为模块未受到攻击。然而，这种技术要求模块必须按照完全一致的特性进行设计，在设计中任何小的变化都会被认为受到后门攻击，而此时模块可能未受到后门攻击。

6. 自动 HDL 代码分析

另一种潜在检测后门的方式是自动分析 HDL（例如 Verilog）代码并标记模块可能的后门。被标记的模块会在运行过程中检测是否有恶意软件 [27]。

表 11-5 对硬件后门攻击种类和一系列阻止攻击的安全策略和机制进行了总结。

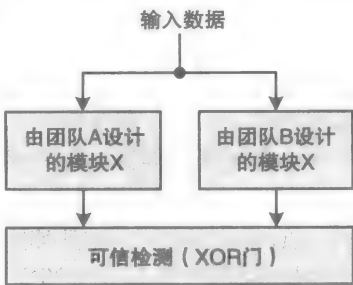


图 11-6 采用模块复制来检测硬件后门攻击 [12]

表 11-5 对硬件后门的安全策略和机制的总结

| 触发类型 | 模块类型 | 安全策略 | 安全机制 |
|------|-------|------------|--|
| 单一输入 | 不可计算的 | 数据混淆 | 使用简单的加密技术。例如，使用随机生成的数字对输入给不可信模块的数据进行按位 XOR 来加密 |
| | 可计算的 | 同态计算 | 使用同态函数 |
| 输入序列 | 不可计算的 | 改变输入顺序 | 对输入数据进行重排或插入虚拟输入 |
| | 可计算的 | | |
| 定时器 | 不可计算的 | 阻止定时器到达目标值 | 对不可信模块进行定期电源复位 |

480

11.3 软件 / 物理攻击

欺骗、拼接、重放和中间人是软件和物理攻击的 4 种类型。可以用来物理地监视计算设备、改变计算设备的功能或者对计算设备进行逆向工程。其中，物理攻击通常被称为硬件攻击，但这不应与之前讨论的硬件后门攻击相混淆。

然而，如果设备采用了适当的具有机密性和完整性特点的安全策略机制，这些类型的攻击都是可检测的。

11.3.1 欺骗攻击

正如前文所述，欺骗攻击是通过向系统中非法插入信息（程序代码或数据）产生攻击，如图 11-7 所示，病毒可以向硬盘或存储器中插入非法数据。拥有与计算设备物理连接的设备也可以使用专门的工具来产生物理欺骗攻击。例如，攻击者可以拦截一次存储器事务并在该存储器中本地修改其数据来欺骗存储器。欺骗攻击破坏了系统数据完整性。

11.3.2 拼接攻击

拼接攻击是通过在系统中非法地让原来信息（指令或数据）与另一信息进行替换，如图 11-8 所示，恶意软件复制一个存储器段的指令到另一存储器段中。这一示例不仅是一个拼接攻击的示例，也是一个物理攻击的示例，其拦截一次存储器事务并向存储器提供其原来访问的地址但数据不同的存储器数据。

即使数据保持机密性，拼接攻击仍可以在不引起注意的情况下发生，攻击者只需要简单

地用已备份存储的数据替代另一部分机密数据而并不需要知道其数据时什么。拼接攻击也破坏了数据完整性，但是采用具有完整性安全策略机制的系统必须集成数据位置信息（例如存储器地址、寄存器号等）来检测这种攻击。拼接攻击的另一个示例是通过替换在页表中存储的指令页号 *A*、*B* 来进行攻击 [29]。在这种情况下，攻击者可以强迫操作系统或进程从物理页号 *B* 而不是物理页号 *A* 处开始执行指令。

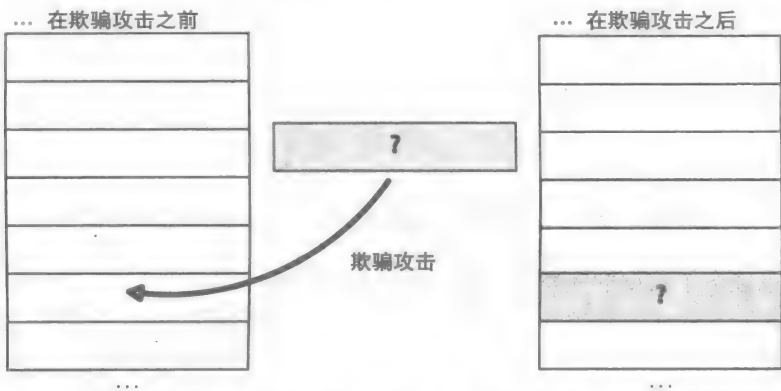


图 11-7 配图说明了一次欺骗攻击 [28]：在目标位置插入了一个不同的值

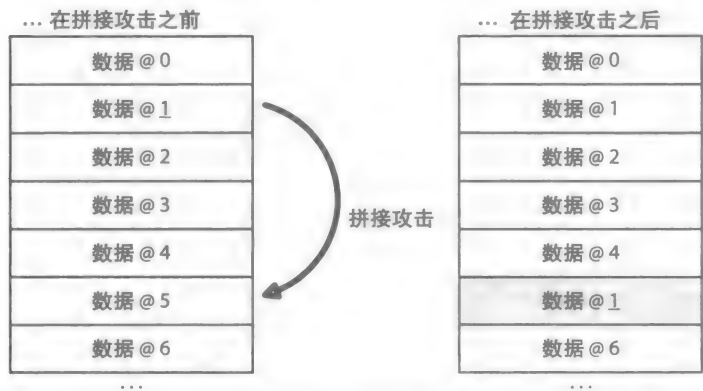


图 11-8 配图说明了一次拼接攻击 [28]：在位置 5 处的数据被位置 1 处的数据替换

11.3.3 重放攻击

重放攻击是在系统中使用当前数据的较早版本非法替代当前数据，如图 11-9 所示。重放攻击与拼接攻击相似，唯一不同之处在于其使用该数据的较早版本而不是系统中的其他数据来替代当前数据。例如，攻击者将一个特殊的数据项放在存储器的某个位置，当系统再次访问该位置时将使用这一特殊数据项。相似的，在硬件层，攻击者可以拦截对地址 *X* 的存储事务，以便保存之后需使用的存储内容复本 [30]。继而攻击者需要等待一次写入存储器 *X* 地址的事务以完成本次攻击。当检测下一次对地址 *X* 的存储器读事务时，攻击者提供之前保存的、相对于现在是旧的内容给处理器。

重放攻击也可能并不是以这样直接的方式来发动攻击的，攻击者可以在系统中同时保存数据的旧版本和新版本。例如，重放攻击可以使用一个虚拟地址映射到两个不同的物理地址，使得新数据存储在一个物理地址，而访问时从存储旧数据的第二个物理地址中获取数据。

481
482

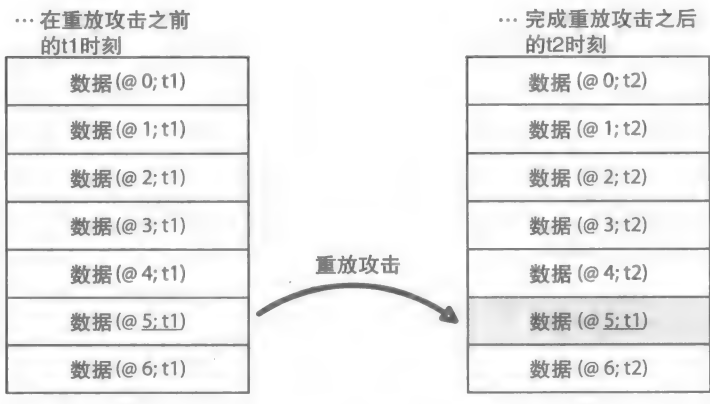
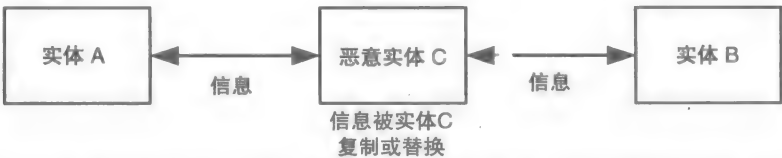


图 11-9 配图说明了重放攻击 [28]: t2 时刻存储在位置 5 的数据被 t1 时刻存储在位置 5 的数据替换了

重放攻击也破坏了数据的完整性，但是以一种不同的方式——所有关于替代数据的事务都是合法的，唯一不同在于数据并不是最新的。例如，在公共事业公司使用智能电表获取用户用电量的过程中，重放攻击可以使得公司每次都读取之前的数据，从而对公司造成一定的损失。重放攻击非常难检测，保持信息的机密性或使用简单的完整性机制都对检测重放攻击不起作用。对于重放攻击的检测，需要使用一种集成了定时信息（如总线事务号、会话话标识符等）的更复杂的安全机制。

11.3.4 中间人攻击

中间人攻击可以发生在以下场景：两个主体（人、软件、固件）并不使用足够机密和完整的安全机制进行通信。一个恶意的主体可以拦截两个合法主体之间的信息交互，并复制主体间通信的消息（如：窃听）或使用不同的信息将最初传送的信息替换，恶意主体担当一个中间人的角色，如图 11-10 所示。实体 A 和实体 B 并未察觉到实体 C 的存在，A 和 B 都以为其与对方直接通信。



483 图 11-10 配图说明了一个中间人攻击。实体 A 和实体 B 均未察觉到实体 C 的存在

11.4 可信计算基

一个 TCB 包括安全设计、开发、安装和运行的硬件和固件以及可能的负责维护系统安全的软件模块。对于硬件，系统必须保护硬件以免其受到后门攻击，从而使得硬件可以正确地运转。这样的硬件被称为可信硬件模块（THM）。相似的，固件也必须被安全地设计、开发和安装，这样的固件被称为可信固件模块（TFM），其是一个嵌入在 THM 中的可防篡改的 IC。此外，在一些使用安全系统应用的区域，一个或多个与安全相关的软件模块也必须安全地执行。在这种情况下，软件模块也必须安全地设计、开发和安装，这样的软件模块被称为可信软件模块（TSM）。

一个 THM-TFM 模块可被组织成一个安全协处理器（SCP），例如，一个作为嵌入式系统的加密处理器可以为操作系统或应用软件提供加密服务。SCP 负责生成安全秘钥（见 11.5 节），安全秘钥是用来保护存储在本地或远程服务器上的文件和数据的机密性和完整性，秘钥也需要安全通信。嵌入在 THM 中的 TFM 适用于保护模块免受欺骗、拼接和重放等攻击，但是其无法防止物理攻击。在 11.10 节本书将使用可信平台模块（TPM）作为 SCP 的示例进行讨论。

另一方面，为了得到最大的适用性，THM-TFM-TSM 也可组织成一个通用目的的安全处理器（SP）。每个 SP 可支持多个安全的执行环境，每个执行环境采用安全执行模式（SXM）来运行任意 TSM。根据应用领域的不同，任意的 TSM 都需要具有机密性或完整性或两者都有的指令，也需要具有机密性或完整性或两者都有的数据。基于 SP 的系统可以支持所有的与安全相关的应用领域，包括那些由 SCP 支持的应用领域。本书给出了一个需要 SP 的安全应用领域示例：当一个商业操作系统被频繁破坏或一个便携式设备需要被保护以免遭受物理攻击时 [31-33]，强制访问控制（11.1.4）的实现。此外，手持设备，如智能电话，也需要一个节能的 SP。例如，考虑数字版权管理策略 [34]，其只允许目标手持设备对加密的媒体文件进行译码。

因为 TSM 可以是软件攻击的目标，其 SXM 必须实现必要的安全策略机制来保护 TSM 免受欺骗、拼接以及回放攻击，若系统是一个便携式设备，也需实现安全策略机制来保护 TSM 免受物理攻击。SXM 将在 11.11 节进行讨论，关于实现最大保护的 SP 的架构将在 11.12 节陈述。

其他需要 TCB 的安全应用领域还包括软件盗版预防、云计算和认证执行等。例如，搜索外星智慧（SETI）的项目 [35] 和在 www.distributed.net[36] 中的利用成千上万志愿用户提供的家庭计算能力并在从公共事业到学术界等众多领域做出重要研究的通用目的的分布式计算项目。用户可以下载一个免费的程序来分析研究来自，如 SETI 项目中的射电望远镜中的数据。然而，如果执行未被认证，是无法验证结果的正确性的。

484

11.5 密码使用方法

机密性是通过应用加密算法，也被称为加密器，来获取并隐藏明码文件、电子邮件、认证数据、存储数据等来实现的。加密器的输出是密文。相似的，使用一个译码算法，也被称为解密器，来对密文进行译码并生成原始的明文，如图 11-11 所示。

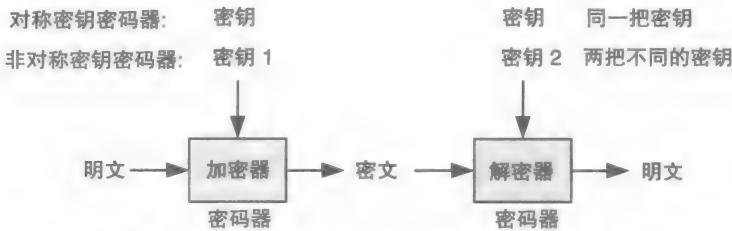


图 11-11 配图说明了加密和解密的使用过程

如果一个加密器生成的密文中不包含任何可以获取到初始明文输入的信息，这个加密器可以被称为是安全的。对称密钥密码器对加密器和解密器均使用同一把钥匙。另一方面，非对称密钥密码器为加密器和解密器使用两把不同的钥匙。在实际的使用中，只需要保密加密

钥匙，而不需要保密加密器或解密器（加密解密算法）[37]。

如果加密器或者解密器每次加密或解密 1 位数据，这样的密码器称为流密码器。此外，如果加密器或解密器每次输入一块数据（多个位），这样的密码器被称为块密码器。当输入信息大于一块时，有多种方式（被称为操作模式）供加密器或解密器选择来加密或解密剩余输入信息。下面将讨论对称密钥密码器和非对称密钥密码器。

11.5.1 对称密钥密码器

图 11-12 中说明了一个 8 位的线性反馈移位寄存器（LFSR），并使用其作为一个简单的对称密钥密码器的示例，这一密码器也被称为流密码器。这一密码器采用包含 4 个调位 4、5、6、8 的平行加载右移寄存器的设计。调位从 1 开始标号，表示寄存器从右向左的寄存器位。通过对 4 个调位对应的位数据进行 XOR 操作从而生成下一个寄存器右移时左侧输入（LI）的数据。

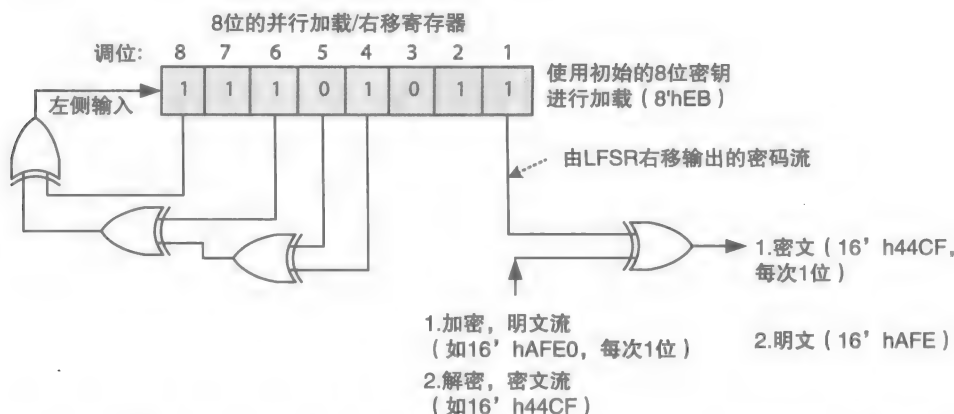


图 11-12 配图说明一个采用调位 4、5、6、8 的 8 位的线性反馈移位寄存器密码器

寄存器使用一个密钥进行初始化，随着寄存器右移，寄存器中的数据将每次输出 1 位从而生成一个密码流。例如，经过 16 次右移，LFSR 将生成 16 位密码流，类似的，也会生成 32 位密码流等。由选择的调位所决定的密码流的周期是密码流重复之前的移位数。在对明文进行加密时，密码流每次与等大小的明文流按位进行 XOR 计算从而生成一个等大小的密文流，每次加密 1 位。相似的，在对密文进行解密时，密码流每次与等大小的密文流按位进行 XOR 计算从而生成相应的初始明文流，每次解密 1 位。

初始化移位寄存器使用的密钥必须小心选择，使得寄存器的内容在右移过程中不会变成 0。在图中，LFSR 密码器使用 8 位 8'hEB（Verilog 中十六进制）来进行初始化，需要 16 个时钟周期来将 16 位明文数据 16'hAFE0 加密成相应的 16 位密文 16'h44CF。相似的，在 16 位密文 16'h44CF 解密之前，需要使用同样的密钥 8'hEB 来初始化寄存器，在 16 个时钟周期中，解密生成初始的 16 位明文 16'hAFE0。

1. A5/1

A5/1 是一种实际使用的流密码器，其使用 19-LFSR、22-LFSR 和 23-LFSR 来进行加密解密过程，如图 11-13 所示。三个 LFSR 分别被标记为 X、Y 和 Z。标记⊕表示按位 XOR。在每个时钟周期中，LFSR 并不进行移位操作，相反，使用寄存器数据 x_8 、 y_{10} 和 z_{10} 作为主

电路的输入，当输入中有两个或两个以上的 1 时，输出 $m = 1$ ，当输入中有两个或两个以上的 0 时，输出 $m = 0$ 。例如，如果 $x_8 = 0$ 、 $y_{10} = 0$ 和 $z_{10} = 1$ ，输出 $m = 0$ ，当 $x_8 = 1$ 、 $y_{10} = 1$ 和 $z_{10} = 0$ 时，输出 $m = 1$ 。

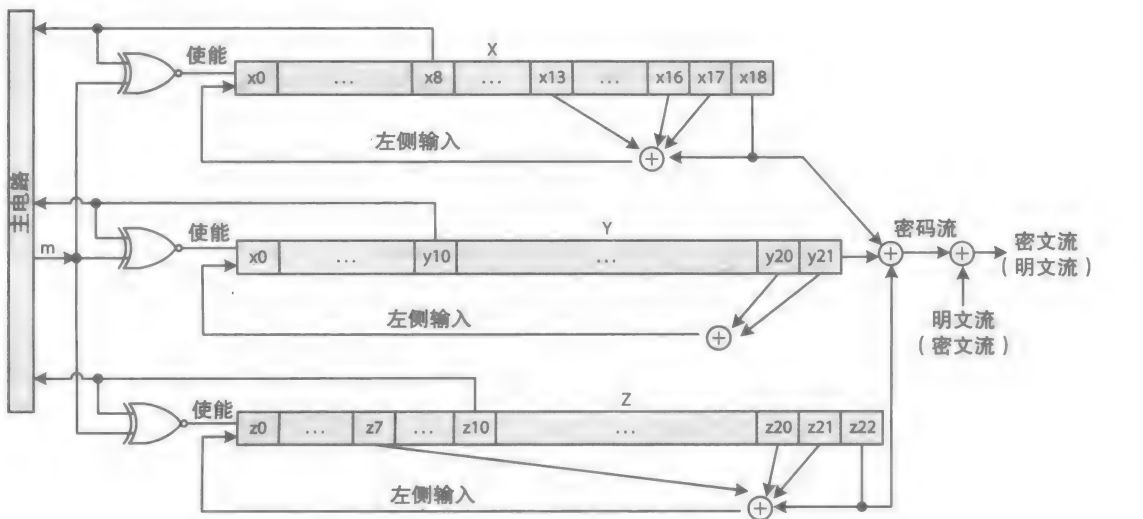


图 11-13 A5/1 流密码器

x_8 、 y_{10} 和 z_{10} 将分别与 m 比较 (XNOR) 来确定在下一个时钟周期是否使用 x 、 y 、 z 对应的寄存器。例如，如果 $x_8 \oplus m = 1$ ，寄存器 X 将被使能，否则，寄存器 X 在下一个时钟周期将不使能。通过以上方式增加了密码流额外的随机性，使得密码流更难被攻击者破解。位 x_{18} 、 y_{21} 和 z_{22} 是用于生成密码流，而最初的密钥有 64 (19 + 22 + 23) 位长。

486

2. 块密码器

数据加密标准 (DES) 是最老的块密码器，其使用 56 位的密钥，操作的块大小为 64 位。而如今最常使用的块密码器是高级加密标准 (AES)，其是由美国国家标准技术研究所 (NIST) [38] 推荐的。

3. 高级加密标准 (AES)

标准的 128 位 AES 密码器要求使用一个 128、192 或 256 位的密钥来加密或解密一个大小为 128 位的数据块。如果明文输入的数据位数量无法被 128 整除，需要在明文的末尾添加额外的位使得其可被 128 整除。AES 将每个输入块按一定的行列顺序组织好，每次加密需要 10、12 或 14 次循环操作，具体为多少次由密钥长度决定。在每次循环中，通过查找表、行移位、列混合以及按位 XOR 等操作生成下一循环输入给寄存器替换其中特定位置的数据。最后一次循环输出的是 128 位的密文。根据操作模式 (接下来讨论) 的不同，解密操作可能以相对于加密操作相反的顺序或相同的顺序执行解密操作。

11.5.2 操作模式

图 11-14 说明了 AES 密码器在密码块链接 (CBC) 模式下的应用。每个块的加密步骤从对当前的 128 位的明文块和与之前生成的 128 位密文块进行按位 XOR 操作开始，正如模式名中的“链接”。然而，对于第一个明文块，其之前并没有生成好的密文块，需要将其与随机生成的 128 位数据进行 XOR 操作，这个 128 位数据被称为初始化向量 (IV)，其并不是必

须保密的。

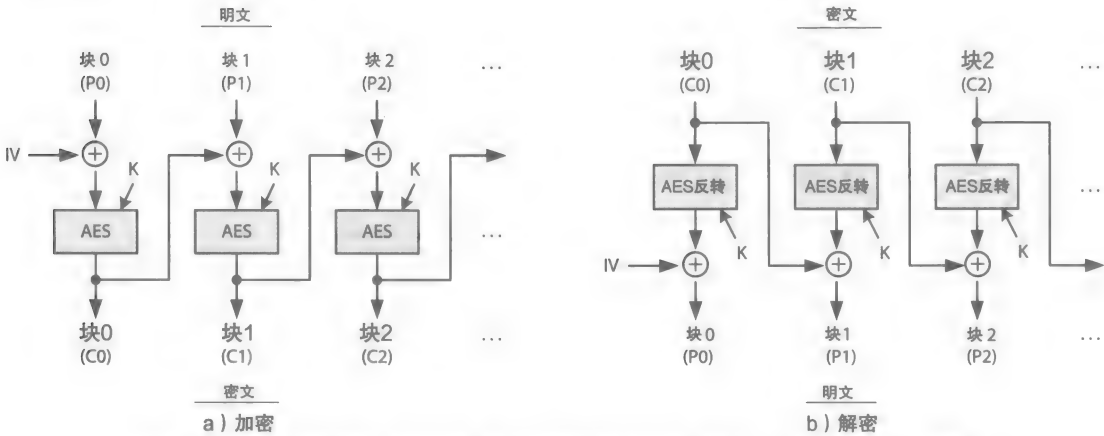


图 11-14 配图说明了在 CBC 模式下工作的 AES 密码器: a) 加密; b) 解密

在 CBC 模式下,对明文块的加密是递归加密的,如图 11-14a 和表 11-6 所示,但是对密文块的解密操作可以以任意顺序甚至是并行的方式进行。在表中,字母 E 表示是加密,字母 D 表示解密,字母 K 表示密钥, $P_0、P_1$ 等表示明文块 0、1 等,字母 $C_0、C_1$ 等表示密文块 0、1 等。

表 11-6 CBC 模式下的加密 / 解密

| 密码块链接模式 (CBC) | |
|------------------------------|------------------------------|
| 加密 | 解密 |
| $C_0 = E(IV \oplus P_0, K)$ | $P_0 = IV \oplus D(C_0, K)$ |
| $C_1 = E(C_0 \oplus P_1, K)$ | $P_1 = C_0 \oplus D(C_1, K)$ |
| $C_2 = E(C_1 \oplus P_2, K)$ | $P_2 = C_1 \oplus D(C_2, K)$ |
| ... | ... |

C: 128 位密文块; P: 128 位明文块; E: 加密器 (如 AES); IV: 初始化向量; K: 密钥; D: 解密器

另一种常用的操作模式是计数器模式 (CTR),如表 11-7 所示。其使用并行处理的 IV 序列来进行加密 / 解密。函数 $E(IV, K)$ 表示使用密钥 K 对 IV 进行加密,再将加密生成的结果与第一个明文块 P_0 进行 XOR 操作从而生成第一个密文块 C_0 。之后 IV 将增加并用于对下一个密文块 (P_1) 进行加密生成密文块 C_1 ,这一加密过程与 C_0 独立不相关。以此类推,完成对所有明文块的加密操作。

表 11-7 CTR 模式下的加密 / 解密

| 计数器模式 (CTR) | |
|---------------------------------|---------------------------------|
| 加密 | 解密 |
| $C_0 = P_0 \oplus E(IV, K)$ | $P_0 = C_0 \oplus E(IV, K)$ |
| $C_1 = P_1 \oplus E(IV + 1, K)$ | $P_1 = C_1 \oplus E(IV + 1, K)$ |
| $C_2 = P_2 \oplus E(IV + 2, K)$ | $P_2 = C_2 \oplus E(IV + 2, K)$ |
| ... | ... |

CTR 模式的优点在于其对加密和解密使用同一个密码器。注意,在表中,并未使用到

解密器 (D)。此外, 对明文块 / 密文块的加密 / 解密操作可并行执行。例如, 一个共 16 块的明文可以被分割为各 8 块的两组明文, 例如, 块 P_0 到块 P_7 为一组, 块 P_8 到块 P_{15} 为另一组。每一组中的块亦可以并行处理。例如, 使用两个线程, 一个线程使用 IV 到 $IV + 7$ 操作块 P_0 到 P_7 , 另一线程使用 $IV + 8$ 到 $IV + 15$ 操作块 P_8 到 P_{15} 。

包括 Intel 和 AMD 在内的多种处理器实现了 AES 指令集, 使得软件中可实现 CBC 模式、CTR 模式和其他一些模式 [39]。

11.5.3 非对称密钥密码器

非对称密钥密码器, 正如前文所述, 需要两把不同的密钥, 一把用于加密另一把用于解密。非对称密钥密码器的最初应用是用于通信, 如例 11-1 所示。在这个例子中, Alice 和 Bob 表示两个主体, 其可以为人、程序或硬件等。

例 11-1 假设 Alice 希望发送密文消息给 Bob。Alice 和 Bob 都含有一个不保密的公共密钥和一个私有的密钥。假定没有其他人发送密文给 Bob, 没有人假扮 Alice, 没有人以任何方式改变 Alice 的消息。

解: 因为 Alice 只关心保持信息的机密性, Alice 和 Bob 可以使用以下步骤:

- 1) Alice 使用 Bob 的公共密钥来加密 Alice 的私有密钥并传送给 Bob。
- 2) 一旦收到消息, Bob 使用其自己私有的密钥对收到的信息进行解密。

因为没有其他人知道 Bob 的私有密钥, 只有 Bob 可以对 Alice 加密的信息进行解密。■

1. RSA

RSA 是使用三位开发这种密码器的人的名字 (Ron Rivest、Adi Shamir 和 Leonard Adleman) 进行命名的。RSA 是一种需要一把密钥进行加密操作并需要另一把不同的密钥进行解密操作的非对称密钥密码器。每一个明文或密文可看作一个整数。例如, 当使用 ASCII 编码的字符串信息 “HELLO”, 其包含 5 个 8 位的 ASCII 码, 其中, 使用 $2'h48$ 或 72 表示字符 H, 使用 $2'h45$ 或 69 表示字符 E, 使用 $2'h4C$ 或 76 表示字符 L, 使用字符 $2'h4C$ 或 76 表示第二个字符 L, 使用 $2'h4F$ 或 79 表示字符 O。

这个字符串可以看作 5 个分离的 ASCII 码, 其对应数字为 72、69、76、76 和 79, 也可看作一个单独的 40 位的大数字 $40'h48454C4C4F$, 也可以使用其他不同的分割方式。例如, 将字符串分割为 3 个 16 位的数字: $16'h4845 = 18501$ 、 $16'h4C4C = 19532$ 和 $16'h4F00 = 20224$ 。最后一个数字是通过在尾部添加 8 位的 0 使得其成为 16 位数字。

为了简化, 假设字符串 “HELLO” 被分割为 5 个明文数字, 为 $P_0 = 72$ 、 $P_1 = 69$ 、 $P_2 = 76$ 、 $P_3 = 76$ 以及 $P_4 = 79$ 。字符串可以使用, 例如之前讨论的 CBC 或 CTR 模式, 进行加密, 每次加密一个数字。然而, 正如之后将要讨论的, 非对称密钥密码器并不适用于大输入数据的加密。

RSA 需要两个互素数作为密钥: 一个作为公共密钥 (e), 用于加密, 另一个作为私有密钥 (d) 用于解密。公式 (11-3) 确定了 n 位明文 P 和 n 位密文 C 之间的关系。

$$\begin{aligned} C &= P^e \bmod n \\ P &= C^d \bmod n \end{aligned} \quad (11-3)$$

假设公共密钥 $e = 5$, 并且 $n = 91$, 公式 (11-4) 说明了密文 $P = 72$ 到对应密文 $C = 11$ 的加密过程, 11 是 72^5 被 91 整除的余数。(注意, 对于 8 位的明文 ASCII 码, n 必须是 256)。

$$\begin{aligned}
 C &= P^5 \bmod 91 \\
 C &= 72^5 \bmod 91 \\
 C &= 1\,934\,917\,632 \bmod 91 \\
 C &= 11
 \end{aligned}
 \tag{11-4}$$

公式 (11-5) 说明了使用私有密钥 $d = 29$ 和 $n = 91$ 对 $C = 11$ 进行解密以获取初始的明文数字 $P = 72$ 的过程。注意, 因为 11^{29} 的值太大, 对于计算器甚至某些计算机都无法计算。其必须被分为更小的形式直到其小到足以被计算。

$$\begin{aligned}
 P &= C^{29} \bmod 91 \\
 P &= 11^{29} \bmod 91; \text{ write } 11^{29} \text{ as } 11^{24} * 11^5 \\
 P &= (11^{24} \bmod 91)(11^5 \bmod 91) \bmod 91; \text{ write } 11^{24} \text{ as } 11^{6*4} \\
 P &= (11^{6*4} \bmod 91)(11^5 \bmod 91) \bmod 91 \\
 P &= ((11^6 \bmod 91)^4 \bmod 91)(11^5 \bmod 91) \bmod 91 \\
 P &= (64^4 \bmod 91)(11^5 \bmod 91) \bmod 91 \\
 P &= (1)(72) \bmod 91 \\
 P &= 72
 \end{aligned}
 \tag{11-5}$$

一个非常大的数 X^y 可以被写成 X^{a+b} 、 X^{a*b} 或 X^{a*b+c} 等形式。公式 (11-6) 说明了 $X^{a*b+c} \bmod n$ 被划分成更小的形式。

$$\begin{aligned}
 X^{a*b+c} \bmod n &= (X^{a*b} \bmod n)(X^c \bmod n) \bmod n \\
 &= ((X^a \bmod n)^b \bmod n)(X^c \bmod n) \bmod n \\
 &= (W^b \bmod n)(Y) \bmod n; \text{ suppose } W = X^a \bmod n, \text{ and } Y = X^c \bmod n \\
 &= (Z * Y) \bmod n; \text{ suppose } Z = W^b \bmod n
 \end{aligned}
 \tag{11-6}$$

例如, 如果公式 (11-6) 中的 W^b 仍然是一个非常大的数字, 其值可以再次划分为更小的形式, 直到其值足够小并对于每个 mod 函数都可计算得出结果。

下面是确定一个加密密钥 e 和解密密钥 d [40, 41] 的算法。公共密钥被安全地存储在一个可信中心, 如存储在公钥密码 (PKI) 中。

- 1) 选择两个素数 p 和 q , 例如 $p = 7, q = 13$ 。
- 2) 确定 $n = p * q$, 即 $n = 7 * 13 = 91$ 。
- 3) 确定 $m = (p - 1)(q - 1)$, 即 $m = (7 - 1)(13 - 1) = 6 * 12 = 72$ 。
- 4) 找到这样的一个素数 e , 需要满足 e 与 m 互素并且 $e < m$; 即 $\gcd(e, m) = 1$ 并且 $e < m$, 其中 \gcd 表示最大公共因子, 例如选择 $e = 5$ 。

5) 找到一个整数 $d = [1 + (k * m)]/e$, 其中 k 是一个整数并且 $d < m$, 例如, 取 $k = 2$, 则 $d = [1 + 2 * 72]/5 = 29$ 。如果没有这样的整数 k , 返回步骤 4) 重新选择不同的 e 值再重复步骤 5), 如果还是没有这样的 e 和 d , 返回步骤 1) 选择另外两个不同的素数, 重复 1) ~ 5) 的操作。

6) 使用值 e 和 n 作为公共 (已知的) 信息, d 作为私有 (不可知的) 信息。即使用 e 作为公共密钥, d 作为私有密钥。

对于以上步骤, 也可以按相反的顺序, 先确定 d 的值, 再根据步骤 4) 和步骤 5) 的要求确定 e 的值。

当 P 、 C 、 e 和 d 是非常大的数字时, RSA 密码器需要进行更多的计算。例如, 一个含有 128 个字母的信息可看作一个 1024 位的整数 ($128 * 8$), 其是由 128 个 ASCII 字符的

$2^{56^{128}}$ 个组合中的一个。这表明 P 和 C 可能是非常大的数字 $< 2^{1024}$ 。对 1024 位的 P 的加密会生成一个 1024 位的密文 C ，其也包含 128 个 ASCII 字符。因为典型的处理器并没有，例如，一个 1024 位的算术单元一共 1024 位 RSA 密码器进行计算，大的算术功能必须通过软件或硬件上的协处理器来进行计算 [42]。

密文 C 和解密密钥 (d) 的值越大，破解 RSA 密码器的难度就越高，其可能需要许多天、月、年来进行计算。一种可能的破解方式是通过蛮力的方式来测试每一个可能的解密密钥，直到确定正确的密钥并得到有意义的结果 P 。然而，例如给一个 2048 位的 C ，其可能需要相当长的时间来确定相应的 2048 位 P 。基于执行解密所需要的时间，可使用定时攻击来排除可能的译码密钥，这就像一个小偷通过判断一个人按预估拨号解锁所需要的时间，来估计解开锁所需要的数字量。

[491]

例 11-2 假设 Alice 想发送一个秘密消息给 Bob，她希望确保没有其他人可以发送消息给 Bob 并且没有人假扮 Alice。

解：

1) Alice 使用其私有密钥对秘密信息进行加密。只要没有其他人知道 Alice 的私有密钥，那么这个加密的信息就有了 Alice 的签名。

2) 之后 Alice 使用 Bob 的公共密钥再次对已经加密过一次的信息再次加密；这样只有 Bob 可以访问该信息，而其他知道 Alice 的公共密钥的人无法访问该消息。

3) 在 Bob 接收到消息时，Bob 首先使用其私有密钥对信息进行解密。

4) 之后 Bob 再使用 Alice 的公共密钥来解密 Alice 发送的信息。 ■

这 4 个步骤要求 RSA 密码器进行 4 次操作：两次由 Alice 完成的加密操作，两次由 Bob 完成的解密操作。考虑到 RSA 密码器加密解密的处理时间，这具有一定的劣势。然而，Alice 和 Bob 可以使用例 11-2 中的步骤来共享一个对称密钥，并使用这个密钥来交换大量数据信息。相对于非对称密钥密码器，对称密钥密码器的加密解密速度要快，并且在实现同样安全等级的天气下，对称密钥密码器需要的密钥位长度也更少；例如，3072 位的 RSA 与 128 位的 AES 实现的安全程度相当 [43]。

例 11-3 假设 Alice 和 Bob 希望交换很多的秘密信息。因此，他们决定使用对称密钥密码器而不是非对称密钥密码器，如 RSA，来进行数据传输，以提升数据传输效率。

解：

1) Alice 和 Bob 使用例 11-2 中给出的 4 个步骤来传输包含如 128 位的对称密钥给 Bob。

2) Bob 和 Alice 都知道了秘密的密钥，他们可以利用这一密钥使用对称密钥密码器（如 AES）来交换秘密信息。 ■

在例 11-2 和例 11-3 中，攻击者有一定的机会实施中间人攻击。例如，假设 Mary 可以访问 Alice 的硬盘或执行定时攻击发现 Alice 的私有密钥。进而 Mary 可以拦截并获取例 11-3 中的 Alice 使用的对称密钥。一旦 Mary 知道了 Alice 和 Bob 间进行通信的对称密钥，Mary 就可以在 Alice 和 Bob 间监控发送秘密信息。因此，需要额外的安全策略机制来检测中间人攻击。

例 11-4 假设 Alice 希望向 Bob 发送对称密钥，但是其希望确保不受到中间人攻击。假定通信媒介是安全的，并且任意的数据传输错误（如果有）都是可被解决的。

解：Alice 和 Bob 需要在其传输的信息中加入对其他人不可知的随机生成的数据，从而确保在其相互通信中没有中间人存在 [44]。在下文中，使用“ptxt”表示明文，使用“ctxt”

[492]

表示密文, “pr”表示私有密钥, “pu”表示公有密钥。

1) Alice 使用 Bob 的公有密钥 (B_{pu}) 来加密一串随机数字 ($r1$), 即 Alice 的姓名或 ID 号 (ID_{Alice}); $r1$ 是由可信软件生成的。Alice 向 Bob 发送该加密信息, 并等待 Bob 回应。

2) Bob 接收到信息后, 使用其私有密钥 (B_{pr}) 来解密信息并发现 $r1$ (标记为 $r1_{Bob-rcvd}$)。

3) Bob 使用 Alice 的公有密钥 (A_{pu}) 来加密并发送包含 $r1_{Bob-rcvd}$ 和另外一串由他的可信软件随机生成的数字 ($r2$) 给 Alice, 发送信息后, Bob 等待来自 Alice 的响应。

4) Alice 接收到消息, 并使用其私有密钥 (A_{pr}) 来解密消息, 从而发现 $r1$ ($r1_{Alice-rcvd}$) 和 $r2$ ($r2_{Alice-rcvd}$)。Alice 将比较 $r1_{Alice-rcvd}$ 和 $r1$ 。如果两个值相同, Alice 可以认为消息由 Bob 发送过来而不是其他人发送的。

5) 继而 Alice 使用 Bob 的公有密钥 (B_{pu}) 加密并发送 $r2_{Alice-rcvd}$ 给 Bob。

6) Alice 也将生成一个明文对称密钥 ($K_{sym-ptxt}$) 并使用 Alice 的私有密钥 (A_{pr}) 将明文对称密钥 ($K_{sym-ptxt}$) 加密成 $K_{sym-ctxt1}$ 。在使用 Bob 的公有密钥 (B_{pu}) 将 $K_{sym-ctxt1}$ 再次加密从而生成 $K_{sym-ctxt2}$, 并发送给 Bob。

7) Bob 使用其私有密钥 (B_{pr}) 对收到的消息进行解密, 从而得到 Bob 发送给 Alice 的 $r2$ ($r2_{Bob-rcvd}$), 通过比较 $r2$ 与 $r2_{Bob-rcvd}$, 若其匹配, Bob 将知道其与 Alice 进行通信。

8) Bob 再使用其私有密钥来将接收到的消息 $K_{sym-ctxt2}$ 解密为 $K_{sym-ctxt1}$, 并使用 Alice 的公有密钥来解密 $K_{sym-ctxt1}$, 从而发现 Alice 的 $K_{sym-ptxt}$ 。

9) Bob 使用 $K_{sym-ptxt}$ 发送确认消息给 Alice。

使用 E 表示加密, 使用 D 表示解密, 使用标记 {} 表示链接, 以上步骤可总结如下:

```
Alice sends:  E( $B_{pu}$ , { $r1$ ,  $ID_{Alice}$ });  $r1$  is a random number
Bob receives: { $r1$ ,  $ID_{Alice}$ } $_{Bob-rcvd}$  = D( $B_{pr}$ , E( $B_{pu}$ , { $r1$ ,  $ID_{Alice}$ }));
Bob sends:    E( $A_{pu}$ , { $r1_{Bob-rcvd}$ ,  $r2$ });
Alice receives: { $r1$ ,  $r2$ } $_{Alice-rcvd}$  = D( $A_{pr}$ , E( $A_{pu}$ , { $r1_{Bob-rcvd}$ ,  $r2$ }));
Alice checks: If  $r1_{Alice-rcvd}$  =  $r1$  then continue; else stop,
               insecure communication.

Alice sends:  E( $B_{pu}$ ,  $r2_{Alice-rcvd}$ );
Alice sends:  E( $B_{pu}$ , E( $A_{pr}$ ,  $K_{sym}$ ))
Bob receives:  $r2_{Bob-rcvd}$  = D( $B_{pr}$ , E( $B_{pu}$ ,  $r2_{Alice-rcvd}$ ))
Bob checks:   If  $r2_{Bob-rcvd}$  =  $r2$  then continue; else insecure
               communication.

Bob receives:  $K_{sym}$  = D( $A_{pu}$ , D( $B_{pr}$ , E( $B_{pu}$ , E( $A_{pr}$ ,  $K_{sym}$ )))));
Bob sends:    E( $K_{sym}$ , message $_{ack}$ );
```

2. 椭圆曲线密码体制

还有其他类型的非对称密钥密码器。椭圆曲线密码体制 (ECC) 在相较于 RSA 提供相同等级的安全前提下要求更小尺寸的公有和私有密钥。例如, 2048 位的 RSA 与 224 位的 ECC 以及 3072 位的 RSA 和 256 位的 ECC[42, 43] 提供相同等级的安全。此外, 因为 RSA 和 ECC 密码器最多有相同大小的密钥需要相同的处理时间, 所以就所需的密钥存储空间和处理时间而言, ECC 效率更高。这使得 ECC 相较于 RSA, 尤其是在需要更少能量的手持设备领域, ECC 更具优势。然而, 从数学上看, ECC 更加复杂。关于 ECC 的描述留待关于密码学的教科书说明。

11.6 哈希法

哈希类似于用于信息确认的指纹。一个哈希函数转换整个信息为一个独特的哈希值, 这个哈希值也被称为哈希码, 或者简单地就称为哈希, 其是少量字节按一定顺序形成的数

字。通过比较接收到的哈希值与从接收到的消息中计算得到的值，来验证消息的正确性，如图 11-15 所示。如果两个哈希值匹配，则认为该消息合法，否则认为该消息或接收到的哈希值或两者均被修改。此外，哈希函数是单方向的，无法通过几字节的哈希值恢复出初始的消息。如果没有秘密钥匙来生成一个哈希，则认为该哈希函数是标准的。

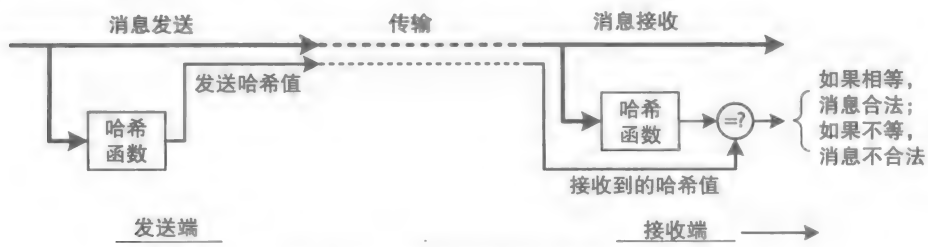


图 11-15 使用哈希进行消息认证

哈希简化了消息确认过程，消息确认可能会使一项艰难的任务，尤其当消息非常大时，如消息为一幅图片或一个二进制文件。图 11-16 说明了一个使用 8 位按位 XOR 逻辑的简单标准哈希函数。在图中，ASCII 字符串“HELLO”生成了一个哈希值 8’h8E。可见，仅仅从哈希值 8’h8E 中是无法确定初始数据“HELLO”的。标准的哈希也被称为校验和，校验和在检测数据传输错误中经常被使用。

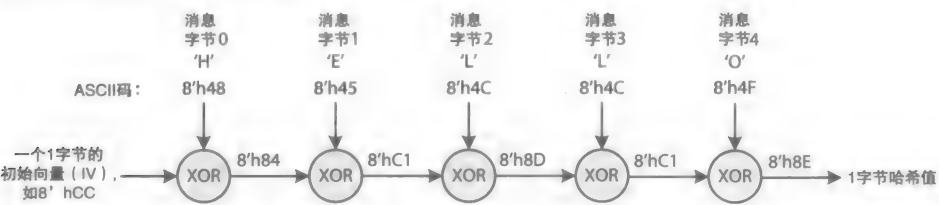


图 11-16 使用 8 位按位 XOR 的简单标准哈希函数。对字符消息“HELLO”其生成了 1 字节的哈希值 494

例 11-5 假设 Alice 希望向其朋友分享一个重要的个人信息，故而她决定在发送消息之前给消息加上数字签名。

解：按以下步骤进行。其中“H”表示哈希函数，“M”表示消息，“HV”表示哈希值：

Alice computes:
Alice signs:
Alice sends:

Friends receive:
Friends extract signature:
Friends check:

$$HV = H(M)$$
$$E(A_{pr}, HV)$$
$$\{M, E(A_{pr}, HV)\}; \text{ note } M \text{ is not encrypted, anybody can read it}$$
$$\{M, E(A_{pr}, HV)\}_{rcvd}$$
$$HV_{rcvd} = D(A_{pu}, E(A_{pr}, HV)_{rcvd});$$
$$\text{If } H(M_{rcvd}) = HV_{rcvd} \text{ then accept } M_{rcvd}; \text{ else reject } M_{rcvd}$$

软件公司也可以使用例 11-5 中说明的技术来安全地分享不保密的软件产品。

哈希函数必须是防冲突的，即两个不同的消息不可以产生相同的哈希值。例如图 11-16 中简单的哈希函数并不是防冲突的哈希函数。如图 11-17 所示，在使用相同的初始向量 $IV = 8'hCC$ 时，字符串“WORLD”和字符串“HELLO”计算得到的哈希值是相同的。

由 NIST[38] 开发的安全哈希算法（SHA）现在包含 4 种哈希算法：初始 SHA-1，SHA-256，SHA-384 和 SHA-512。对于字符串“HELLO”和“WORLD”，SHA-1 生成了下列两个哈希值 [45]。这两个哈希值是非常不同的。然而，SHA-1 也并不是防冲突的 [40]。

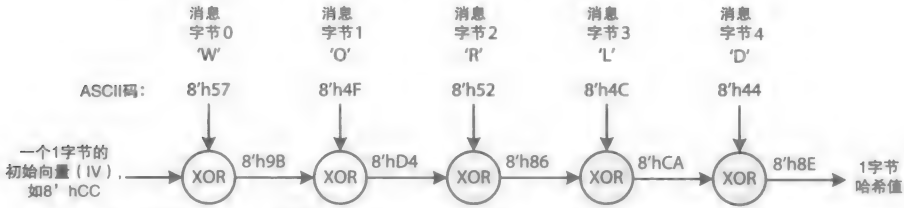


图 11-17 通过实例说明哈希函数是不防冲突的，两个消息——“WORLD”和“HELLO”——生成了相同的哈希值

消息: “HELLO”
SHA-1hashvalue:c65f99f8c5376adadddc46d5cbcf5762f9e55eb7

消息: “WORLD”
SHA-1hashvalue:1a5db926797b9ae16ad56ec2c143e51a5172a862

495

每一个哈希算法成功地生成了一个 512 位（SHA-1 和 SHA-256）或 1024 位（SHA-384 和 SHA-512）信息块，并最终分别生成一个 160、256、384 或 521 位的哈希值。在一个信息进行哈希之前，必须添加额外的位（如果需要），再与表示消息大小的数进行连接。通过添加并连接生成的输入消息的长度必须是块大小的整数倍。例如，图 11-18 说明了使用 SHA-512 算法的输入消息格式。先在消息末尾添加一定数量的 0，再将其与使用 128 位的无符号整数表示的消息长度 N 连接从而生成最终的长度为 $M * 1024$ 位的消息，其中 M 为整数。表 11-8 列出了各种 SHA 算法的相关特性。

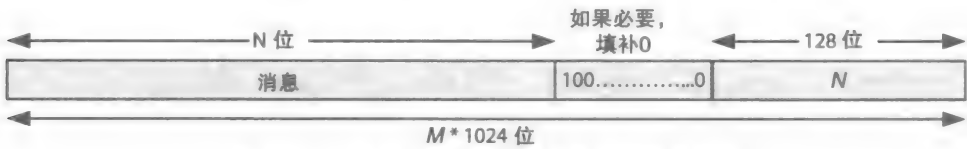


图 11-18 使用 SHA-512 哈希算法的消息格式

表 11-8 安全哈希算法特性

| 特 性 | SHA-1 | SHA-256 | SHA-384 | SHA-512 |
|--------------|------------|------------|-------------|-------------|
| 哈希值大小 (位) | 160 | 256 | 384 | 512 |
| 块大小 | 512 | 512 | 1024 | 1024 |
| 消息大小 (明文或密文) | $< 2^{64}$ | $< 2^{64}$ | $< 2^{128}$ | $< 2^{128}$ |
| 步数 | 80 | 64 | 80 | 80 |

如图 11-16 中简单的哈希函数，一个 SHA 以 k 位的 IV 为开始，一般情况下 k 小于块大小 n ($k < n$)，再经过多次压缩从而生成第一块的哈希值，这一哈希值将作为下一块的 IV。继续以上过程，直到生成最终的哈希值，这一哈希值也被称为消息摘要。对消息的任意改动都会导致生成一个不同的哈希值，使得其无法匹配伴随初始消息一起传送的哈希值。SHA 算法既可以用于明文输入，也可以用于密文输入。

11.7 加密哈希

一个标准的哈希是不能被保护的。而且还有可能让对手同时改变消息值和它的哈希值而不被发现。另一方面，加密哈希算法一个密钥来产生一个哈希值。在这种情况下，这种哈希

被称为基于关键字的哈希。下面将介绍基于关键字的哈希算法的两个实例。

11.7.1 消息认证码

消息认证码 (MAC) 也叫加密 MAC (CMAC), 需要密码来生成一个安全哈希。例如, 图 11-19 表示了使用 n 位块的 AES-CBC 加密器和一个 k 位的密钥 (K) 生成 MAC 的过程。图中 n 位的密码 $K1$ 和 $K2$ 取决于消息的大小和一个常数, 其中消息的大小由 K 确定, 常数取决于 $n[40]$ 。如果消息的大小是一个由 n 整除的整数, 那么将使用常数 $K1$; 否则, 最后的块要被填充 1 和 0 来创建一个 n 位的最后的块, 然后常数 $K2$ 要被用到。最后的加密块中的几字节被选择作为 MAC。

496

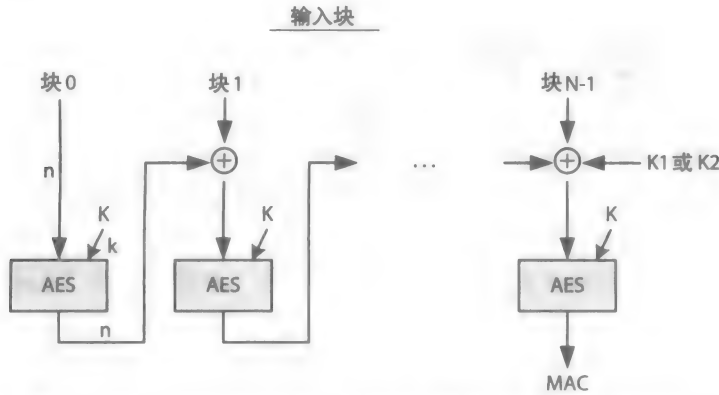


图 11-19 AES-CBC-MAC; MAC 长度为 m 位, 其中 $m \leq n$

这个组合缩写 AES-CBC-MAC 表示用来产生 MAC 的 CBC 模式中的 AES 密码。AES-CBC-MAC 有一个优点就是能够同时实现加密和基于关键字的哈希。

如果在一个或多个输入块中有意外或手动的改动, 这个算法产生一个不同的 MAC。因此, 对方如果不知道密钥而去改变输入, 仍能产生相同 MAC 的事情是不可能的。然而, 发送者和接收者都需要知道这个密钥, 这种唯一标识用来产生一个 MAC。另外, MAC 作为一个明文或密文的输入而产生。

11.7.2 基于哈希的 MAC

与 CMAC 需要一个密码相对比, 基于哈希的 MAC (HMAC) 需要一个更有效 (计算量更少) 的标准哈希算法, 如 SHA-256。HMAC 要在两个散列周期中使用两个额外的密 (S) 码 S_i (输入 S) 和 S_o (输出 S) 来计算而得, 这两个密码由一个密钥 K 和两个整数常数而得, 两个整数常数分别叫作输入摘要 ($iPAD$) 和输出摘要 ($oPAD$)。

$iPAD$ 和 $oPAD$ 本身不会使这个哈希算法更安全; 相反, 它们通过一个被称为白化的技术来提高密钥的质量 [46]。例如, 给出一个初始 16 位的密钥 = $16'h1234$, $iPAD = 8'h36$, 一个由作为白化技术的按位 XOR (\oplus) 产生的 32 位的 S_i :

497

K: 由 0 产生的原始密钥 00000000 00000000 00010010 00110100 (0x00001234)

重复的 $iPAD$ ($iPAD^+$): 00110110 00110110 00110110 00110110 (0x36363636)

\oplus

32 位 S_i : 00110110 00110110 00100100 00110010 (0x36362432)

(11-7)

给出消息 M , 密钥 K , $iPAD$ 和 $oPAD$ 的值, 一个标准的哈希算法, HMAC 的产生如下。

HMAC 算法:

1) 如公式 (11-7) 所示, 由 K 和 $iPAD$ 产生 Si ; 即 $Si = K \oplus iPAD^+$ 。原始密钥可以由摘要 0 产生, $iPAD$ 需要被重复来产生所需长度的 Si 。

2) 由 Si 和 M 拼接产生哈希值 (HV); 即 $HV = H(\{Si, M\})$, 其中 $\{\}$ 表示拼接。

3) 通过 K 和 $oPAD$ 产生 So (即 $oPAD = 8h'5C$)。 So 由与 Si 相似的方式产生; 即 $So = K \oplus oPAD^+$ 。

4) So 拼接 HV 产生 $HMAC$; 即 $HMAC = H(\{So, HV\})$ 。

公式 (11-8) 总结了这 4 个步骤。

$$HMAC(K, M) = H(\{K \oplus oPAD^+, H(\{K \oplus iPAD^+, M\})\}) \quad (11-8)$$

11.8 通过硬件存储加密密钥

正如本章开头所讨论过的, 保密性和完整性对计算机安全而言是至关重要的。然而, 在硬盘上存储加密密钥对系统构成安全隐患和威胁。安全存储许多密钥, 比如在一个组织中用到的这些, 需要在防篡改的 IC 内部创建一个绑定保密密钥的密钥结构。这被称为通过硬件或绑定数据到平台的数据存储。

NIST 给出了一组推荐的密钥大小和推导技术 [38]。例如, 推荐了 2048、3072 或 4096 位密钥大小的 RSA, 或 256、384 位密钥大小的 ECC 来对公共密码加密。类似地, 推荐了 256、384 和 512 位的 SHA 进行哈希。AES-CBC-MAC 是保护保密性和完整性的推荐之一。
[498] 仅为了完整性, 推荐使用 SHA-256 的 128 位 HMAC。读者参考 NIST 文件可以得到应用程序的具体建议。加密密钥的安全存储必须由一个可信计算基 (TCB) 来维持。

11.8.1 密钥链组织

一个密钥结构, 或者说密钥链, 通过父亲和叶子结点分层次地组织为一棵树。图 11-20 表示了一个或多个密钥链的组织示例, 每一个带有一个或多个结点 [47, 48]。在这个图中, 父亲结点用方框表示, 它与保护孩子结点值的存储密钥有关。叶子结点, 表示为三角形, 与签字密钥有关, 也被称为签名密钥。例如, 它被用来加密邮件信息的散列, 或程序验证结果的输出。表示为圆形的叶子结点表示了少量的数据, 例如, 加密大数据文件的对称密钥。一个密钥结构也包括其他叶子结点, 比如身份证明密钥 (AIK)。它是一个绑定平台, 而且比如会用在服务器中验证身份的应用程序的非对称密钥。

根父亲结点由一个安全存储在防篡改 IC 内部的加密根键 (SRK) 保护起来。保密密钥基于物理防克隆技术 (PUF), 这种技术已被证实可以抵抗多种类型的攻击 [49-51]。一个箭头表示一次密钥的推导。每一个父亲和叶子结点包括一系列的密钥资料, 其中包含一个被称为随机数 (只使用一次的数字) 的特殊数字, 而且也可能包括一个或多个确定密钥类型的常数。这些密钥本身没有被保存; 只有每个结点的密钥资料被安全地保存了下来。另外, 在一个孩子密钥可以被使用之前, 授权用户通常需要提供一个或多个正确的密码, 它们也被称为授权数据。密钥链可以在多种方式中被构建 [48, 52-55], 比如接下来将要讨论的这些。

11.8.2 存储和访问

图 11-21a 表示了一个含 10 个结点的密钥链, 其中有 4 个父亲结点, 分别标识为 0、2、

5 和 7, 还有 6 个叶子结点, 分别被标识为 1、3、4、7、8 和 9。随机数 N_0 被分配到根父亲结点 0, N_1 被分配到数据叶子结点 1, N_3 被分配到签名叶子结点 3, 等等。如图 11-21b 所示, 每个结点生成的记录被本地或远程地安全保存在服务器中。每条记录都包含了一个密钥标识号, 例如, 结点号; 父亲标识号; 一个密码和哈希算法的名字, 称为密码算法的标识符; 加密密钥资料; 还有整个记录的加密哈希。

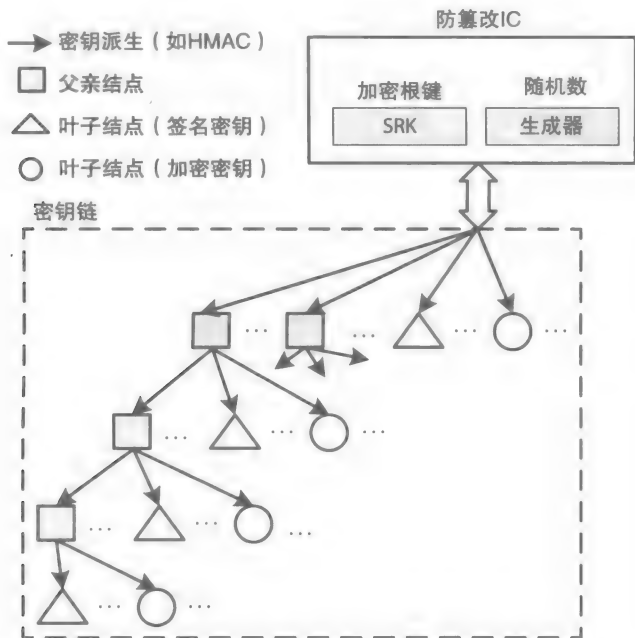
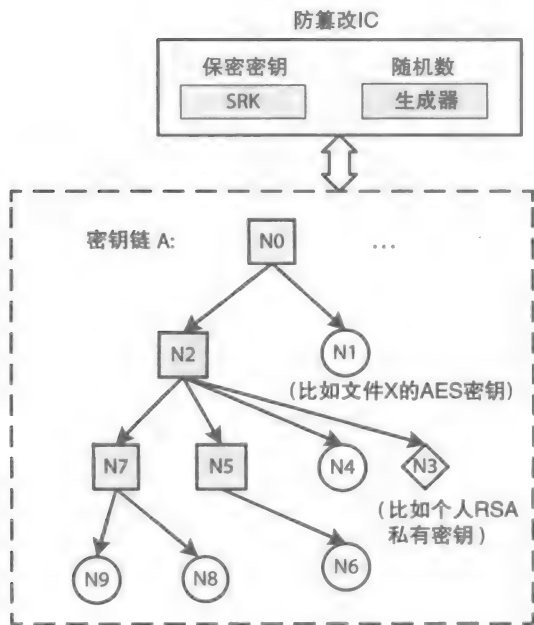


图 11-20 绑定防篡改 IC 内部 SRK 的密钥链



a) 一条保密密钥

| |
|-----------------------------|
| <p>结点标识号 (ID)</p> |
| <p>父亲标识号 (PID)</p> |
| <p>密码算法标识符 (CAls)</p> |
| <p>加密密钥资料 (EKM)</p> |
| <p>记录基于关键字的哈希 (RHV)</p> |

b) 一条结点记录

图 11-21 一个密钥结构示例: a) 密钥链; b) 作为一条记录的结点 [53, 55]

499
500

SRK 是嵌入每一个 SCP 或 SP 芯片的安全唯一密钥。钥匙可以在安装过程中通过不干预操作系统的一套安全的 I/O 机制被编程到芯片 [55]。在这个图中，根父亲结点 0 的密钥被用来保护它的两个孩子结点 1 和 2；数据叶子结点 1 的密钥被用来保护比如一个 128 位的 AES 密钥，该密钥被用来加密一个较大的用户文件或一个较大的应用数据；父亲结点 2 的密钥被用来保护它的 4 个孩子结点 3、4、5 和 7；等等。

密钥链通过 SCP 中受信任的固件模块 (TFM) 或通过运行在 SP 上可信软件模块 (TSM) 来维持 (也可参考 11.4 节)。TFM/TSM (TFM 或 TSM) 将包含一系列的应用程序接口 (API)，比如操作系统所使用的 “Add2Keychain” 和 “Encrypt”，或一个应用程序请求的安全加密服务。

在以下示例中，假设每一个结点可以使用不同的加密和哈希算法。另外，为了简化，不需要授权数据。明文标注 (ptxt) 的所有项目都被认为是 SCP 或 SP 中安全的。这个示例假设一个应用正在请求 SCP 或 SP 安全服务。

例 11-6 应用软件请求 SCP 的 TFM 或 SP 的 TSM (表示为 TFM/TSM)，来将父亲结点 0 添加到图 11-21 中的密钥链 A 上。

解：

应用程序任务：

使用以下的 API，发送一个请求到 TFM/TSM 来添加根父亲结点 0 并创建一个密钥链；TFM/TSM 将使用应用程序提供的 cipher0 和哈希算法 algHash0 来加密并生成一个密钥资料的基于关键字的哈希。根结点 0 的父亲是空。

```
Addkey2Keychain(keychain_A, 0, Null, cipher0, algHash0, nodeType0, R0) //nodeType: parent
```

TFM/TSM 任务：

生成一个随机数，然后使用 SRK 来加密这个随机数和结点 0 的密钥资料。记录也使用 SRK 进行基于关键字的哈希。隐藏记录返回到应用程序存储下来。下面，“结点类型”表示父亲或叶子（数据）结点， R 表示在存储器中的密钥记录数据结构的引用，“KID”表示一个密钥结点 ID（一个数字），“PID”表示一个父亲结点 ID（一个数字），“EKM”表示加密密钥资料， N 是一个随机数， HV 是一个哈希值。SRK 是安全根密钥，如果使用 TFM 实现这些 API，它就存储在 SCP 芯片中，或者如果使用 TSM 实现这些 API，它就保存在 SP 芯片中。特别地，TFM/TSM 使用 SRK 来完成以下的任务，因为 $KID = 0$ ， $PID = \text{null}$ ：

```
Addkey2Keychain(keychain, KID, PID, cipher0, algHash0, nodeType, R)
```

begin

```
 $N_{ptxt} = \text{nonce}();$  //generate a nonce for root Node 0
```

```
 $EKM_{ctxt} = E_{cipher0}(SRK, \{N_{ptxt}, nodeType\});$  //encrypt key material using SRK since PID=null
```

```
 $HV = H_{algHash0}(SRK, \{KID, PID, cipher0, algHash0, EKM_{ctxt}\});$  //keyed hash of the record
```

```
 $R = \{KID, PID, cipher0, algHash0, EKM_{ctxt}, HV\}$ 
```

end

例 11-7 应用软件请求 TFM/TSM 来添加数据叶子结点 1 到图 11-21 中密钥链 A 的父亲结点 0 上。

解：

应用程序任务：

501

使用以下的 API, TFM/TSM 被指示添加一个加密密钥结点到密钥链 A 的父亲结点 0 上; 应用程序提供 cipher1 和 algHash1 来对结点 1 的密钥资料进行加密和基于关键字的哈希, 其中 K 表示一个密钥 (也可以参考 11.6 节)。

```
AddKey2Keychain(keychain_A, 1, 0, cipher1, algHash1, nodeType1, R1); //node type: data
```

TFM/TSM 任务:

完成以下的操作, 因为 PID = 0:

```
R0 = getParentRecord(keychain_A, 0); //0, Null, EKM0ctxt, cipher0, algHash0, HV0read
HV0' = HalgHash0(SRK, {0, Null, cipher0, algHash0, EKM0ctxt}); //re-compute hash
HV0' = ? HV0read if yes, continue; otherwise, raise an exception //authenticate Node 0
EKM0ptxt = Dcipher0(SRK, EKM0ctxt); //EKM0 contains N0ptxt
K0:1 = Ecipher0(SRK, N0ptxt, 1, nodeType1); //generate key for Node 1;
N1ptxt = nonce(); //generate a nonce
EKM1ctxt = Ecipher1(K0:1, {N1ptxt, nodeType1}); //encrypt Node 1's key material
HV1 = HalgHash1(K0:1, {1, 0, cipher1, algHash1, EKM1ctxt}); //hash key material
R1 = {1, 0, cipher1, algHash1, EKM1ctxt, HV1} //Record is returned to application
```

例 11-8 应用程序软件请求 TFM/TSM, 并使用图 11-21 中第一个数据叶子密钥来对应用程序数据进行加密。

解:

应用程序任务:

使用以下的 API, TFM/TSM 被指示使用密钥链 A 中的加密密钥 1 来对应用数据进行加密, 其中 $data_{ptxt}$ 和 $data_{ctxt}$ 引用内存中的应用程序数据结构。(也可以参考例 11-6 和例 11-7)

```
Encrypt(keychain_A, 1, 0, dataptxt, application_cipher, datactxt); //use application cipher
```

TFM/TSM 任务:

使用数据叶子结点 1 的密钥资料产生加密密钥 1, 然后使用应用程序提供的密码对应用数据进行加密。应用程序数据作为密文返回到应用程序中。

```
R0 = getParentRecord(keychain_A, 0); //0, Null, cipher0, algHash0, EKM0ctxt, HV0read
HV0' = HalgHash0(SRK, {0, Null, cipher0, algHash0, EKM0ctxt}); //recompute hash
HV0' = ? HV0read if yes, continue; if no, raise an exception
EKM0ptxt = Dcipher0(SRK, EKM0ctxt); //will use N0ptxt
K0:1 = Ecipher0(SRK, {N0ptxt, 1, nodeType1}); //Key for Node 1;
R1 = getRecord(chain_A, 1); //1, 0, cipher1, algHash1, EKM1ctxt, HV1read
HV1' = HalgHash1(K0:1, {1, 0, cipher1, algHash1, EKM1ctxt}); //recompute hash
HV1' = ? HV1read if yes, continue; if no, raise an exception
EKM1ptxt = Dcipher1(K0:1, EKM1ctxt); //will use N1ptxt and nodeType1
Kdata-leaf = Ecipher1(K0:1, {N1ptxt, nodeType1}); //generate the encryption key
datactxt = Eapplication_cipher(Kdata-leaf, dataptxt); //encrypt data
```

表 11-9 表示了图 11-21a 中结点 0 ~ 9 的假想记录; 只有 4 条记录表示了出来。这个图中的事实是, 密钥链父亲结点 0 被 SRK 隐藏在 IC 中; 结点 1 和 2 被父亲结点 0 隐藏起来; 结点 3、4、5 和 7 被结点 2 隐藏起来; 结点 6 被结点 5 隐藏起来; 结点 8 和 9 被结点 7 隐藏起来, 这个密钥链可以说是被硬件密封起来。

表 11-9 使用图 11-21b 中格式表示的图 11-21a 中的记录密钥链

| 密钥标识号 (KID) | 父亲标识号 (PID) | 加密算法标识符 (CAI) | 加密密钥资料 (EKM) ¹ | | 整个记录的基于关键字的哈希 (RHV) ² |
|----------------|----------------|-------------------|---------------------------|-----------|-------------------------------------|
| | | | 随机数 ² | 随机数类型 | |
| 0 | null | cipher0, algHash0 | 6564...465 | parent | 3A5A...C20B |
| 1 | 0 | cipher1, algHash1 | 8815...489 | data | 1ECA...C360 |
| 2 | 0 | cipher2, algHash2 | 6304...363 | parent | 45F6...2BEC |
| 3 | 2 | cipher3, algHash3 | 2467...427 | signature | 1F55...F8B9 |
| ... | ... | ... | ... | ... | ... |

1. 非加密密钥资料表示；2. 模拟值

11.8.3 应用示例：密钥链作为访问控制

一个结点离密钥链的根越远，越需要更多的计算量来确定一个密钥。因此，使用分层的授权数据访问方案，一个密钥链可以被用来实现多级的访问控制（11.1.4 节）。图 11-22 表示了一个有三个部门的公司数据组织。在每一个部门中，数据被分类到几个安全级别中。例如，在工程部门中，数据被分为 4 类，它们分别被工程师、产品工程师、项目经理和主任访问。例如，任何作为普通和特权“工程师”的人都有权访问被分类为“工程数据”的所有数据。产品工程师有权访问所有的“产品数据”以及所有的“工程数据”。然而主任有权访问“指导数据”，以及他们部门的其他所有数据。

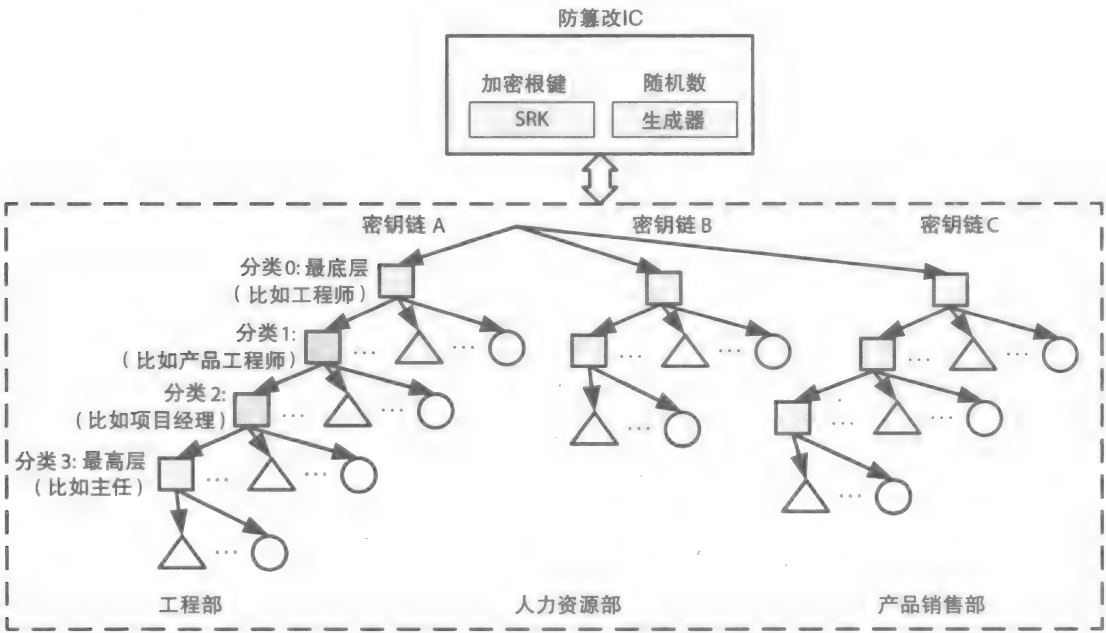


图 11-22 一个访问控制密钥链 [54]

密钥链通过使用 SRK 所保护的每类数据（图 11-22 中有 4 类）需要的 RSA 公有 / 私有密钥对和每个结点需要一个随机素数被组织起来 [54]。使用分类 j 到分类 0 通道中的所有公共密钥，可以计算出每一类 $j \geq 0$ 的数据的初始授权数据。为了访问分类 j 中的一个叶子结点 k ，使用分类 j 中指定的最初的授权数据，分类 j 到分类 0 通道中的所有私有密钥，还

有指定到结点 k 的随机数，会得到一个授权数据。

因此，一个授权数据的计算——例如最底层分类“工程数据”——将仅包含一个私有密钥，由于顶层分类“指导数据”的授权数据的计算需要用到 4 个私有密钥，会使“指导数据”更安全。

11.9 哈希树

尽管图 11-21 中的密钥链可以被硬件隐藏起来，但其仍然可以受到重放攻击，因此是不安全的（11.3.3 节）。恶意软件可以保存整个密钥链，等到密钥链更新时，发起重放攻击。这样，它用更新的密钥链代替之前存储的密钥链，从而可以阻挡对隐藏数据的访问且可以使合法用户无法进入系统。除了使用存储在防干扰的集成芯片中的哈希密钥链，没有其他方法可以检测到密钥重放攻击。每一次密钥链中的密钥被用到时，密钥链进行更新，哈希树需要重新计算。给定的密钥链有时候很大，包含上千个密钥，除非利用哈希树，否则这将是一项昂贵的任务，在之后举例中将会讨论。

11.9.1 应用示例：密钥链认证

图 11-23a 展示了图 11-21a 中的密钥链对应的哈希树，也称为默克尔哈希树 [56]。图中箭头有两个指向——从父结点指向子结点的方向表示生成密钥，从子结点指向父结点的方向表示生成哈希。每一次密钥链更新或者密钥被使用时，不是将整个哈希树重新计算一遍（密集型任务），而是计算父结点的子结点哈希然后存储在父结点中。根结点哈希，也叫作安全根哈希（SRH），存储在 IC 中，如图所示。

图 11-23b 中展示了用于父结点的记录结构。在此例中，每条父结点记录包括其子结点记录的哈希值，而不是记录哈希值（RHV）（图 11-21b）。图 11-24 展示了用模拟值作为结点内容的密钥链哈希树示例。图中 8 位按位异或作为哈希函数。假设图中的每个父结点包含附加 16 位内容，如图 11-23b 所示，用于从其子结点内容中计算哈希值。叶结点只有 16 位数据而没有哈希值。

图 11-24a 展示了对初始哈希树根哈希值 $8'hC0$ （Verilog 的十六进制表示）的计算过程。哈希值作为一个 SRH 存储于芯片中。在图 11-24b 中，子结点的内容从 $16'h2345$ 变为 $16'h2355$ ——只有一位变化。由此产生了新的根哈希值 $8'hD0$ 。如果这个变化是由正常的更新引起的，那么 $8'hD0$ 将代替 $SRH = 8'hC0$ 。另一方面，如果这个结果是由一次攻击引起的，这个变化将会被检测到并且 $8'hD0$ 不会匹配 $SRH = 8'hC0$ 存储到集成芯片中。

11.9.2 应用示例：内存认证

默克尔哈希树还有别的应用，例如图 11-25 中所示用于内存认证的 n 元哈希树。TSM 的完整性可以通过验证其在内存中的指令和数据来保护。每一个结点都有 n 个子结点。当 $n = 2$ 时，对应哈希树是二叉树。整个虚拟或者物理内存，或者一部分，都可以被组织成用每个结点内存块构成的 n 元哈希树。叶结点块包括指令或者数据，而父结点块只包含从子结点块计算的哈希值。

图 11-26 展示了一棵用 4 个数据块作为叶结点和 3 个哈希块作为父结点的二叉哈希树。在图中，假设每块的大小都为 2 字节（2B），且 8 位按位异或作为哈希函数。每个父结点块存储两个 8 位哈希值，从其两个子结点块获得。根结点块的哈希值作为 SRH 存储在 IC 中。

在图中，原始 $SRH = 8'h00$ 。每个结点块的改变（包括父结点和叶结点）都会导致根哈希值的改变。注意到结点内容的改变只引起从其到根结点路径上结点内容的改变。这样，当只有一个结点块更新之后只有一小部分的结点块受到影响。

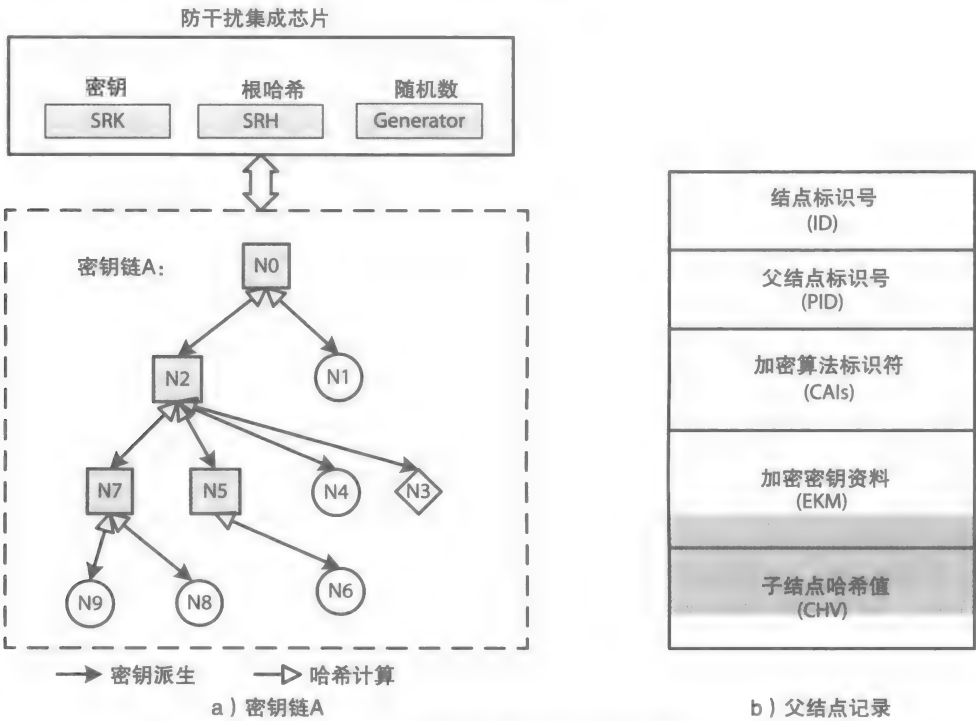


图 11-23 密钥链的默克尔哈希树表示

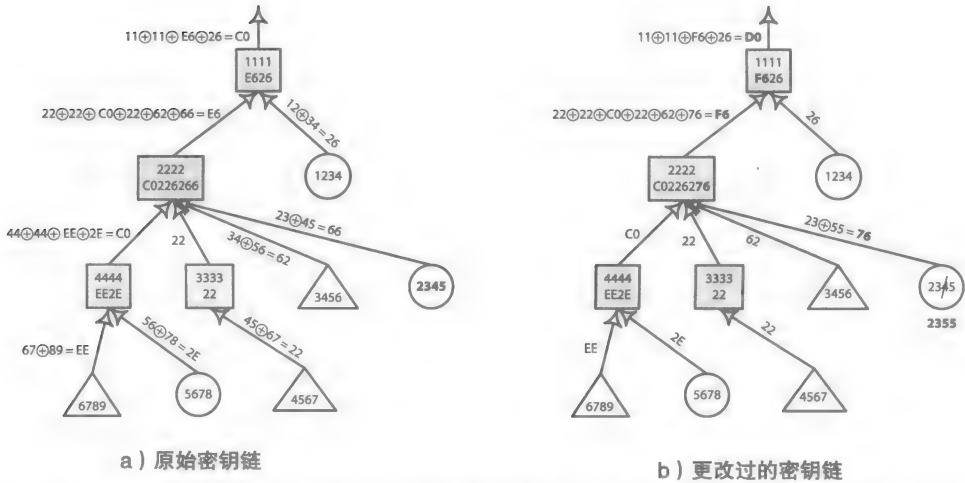


图 11-24 用 8 位按位异或作为哈希函数计算根哈希：a) 原始哈希树，根哈希值 = 8'hC0；
b) 被更改过的哈希树，根哈希值 = 8'hD0

组织 n 元哈希树有很多种方法。例如，有 32B 节点块的哈希树可以用有 1B 哈希值的 32 元哈希树组织，或者用有 2B 哈希值的 16 元哈希树组织，或者用有 4B 哈希值的 8 元哈

希树组织，或者用有 16B（128 位）哈希值的二叉哈希树组织。此外，父结点块可以和叶结点块存储于同一块内存或者分开存储。

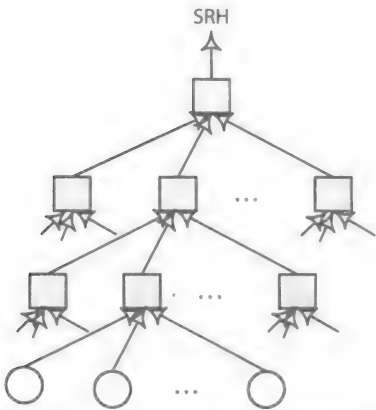


图 11-25 用叶结点（圆圈）表示数据和父结点（正方形）表示哈希值的 n 元哈希树

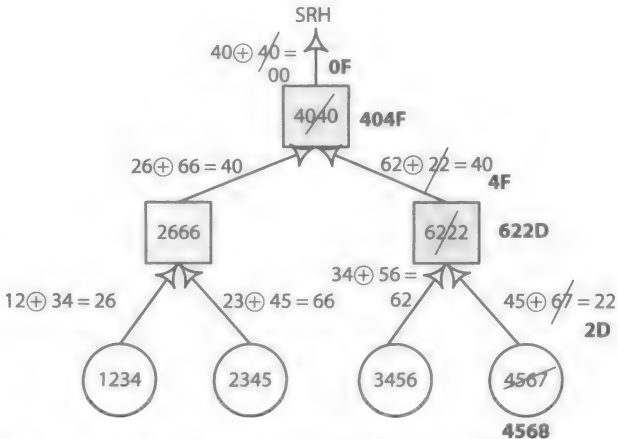


图 11-26 一个叶结点内存块改变的 二叉内存认证哈希树

11.10 安全协处理器体系结构

如之前在 11.4 节中讨论的一样，SCP 作为嵌入式系统，其包含一个可信固件模块（TFM）。因为固件的指令和数据不能从芯片外部直接读取，TFM 不受欺骗、拼接或者重放攻击的影响。然而，由于 SCP 必须要和平台中其他部分交换数据，如果攻击可以控制平台物理结构，那么 SCP 也有可能遭受物理攻击。例如，在从内存访问 SCP 的任何数据都是不安全的。

所需的加密算法可以作为 TFM 的一部分在软件中实现，但是运行起来会需要很多时间，所以，为了提高效率，加密算法一般在硬件中实现。图 11-27 展示了包含所需模块最小集的 SCP 结构。其包括存储加密根键（SRK）的非易失存储器、随机数产生器和硬件中实现的加密 / 解密哈希算法。SRK 可以是对称密钥 [53] 或者公共 / 私有密钥对中的私有密钥 [47]。

505
507

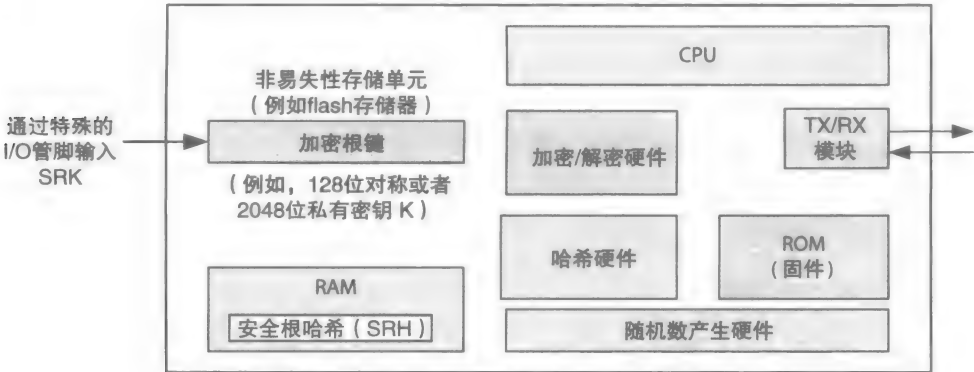


图 11-27 嵌入式系统中的安全协处理器

在 TFM 执行和临时将密钥存储在密钥链中（参见 11.8.2 节）时，会用到随机存取存储器（RAM）。如果 TFM 也管理被哈希树保护的密钥链，那么 RAM 也可以用于存储 SRH。

SCP 的例子有智能卡 [47] 和可信平台模块 [TPM][47, 52, 58]。然而，智能卡只有很小的存储空间。

可信平台模块

TPM 用于提供整个平台的安全；其保证了硬件和操作系统软件部分在启动时的有效性。操作系统甚至其他应用程序可以使用 API 去访问 TPM 的安全服务。

TPM 是由被称为行业代表的可信赖计算集团（TCG）开发的产品。许多公司，包括 AMD、HP、IBM、Intel 和微软，都是 TCG 的成员。图 11-28 展示了嵌入式系统中 TPM 芯片的框图。非易失性存储器用于存储签注密钥（EK）、SRK 和启用或禁用某些功能的标识。EK 是一种嵌入在芯片中的密钥，通常由厂商写入。SRK 用于保护 TPM 生成的密钥。身份证明密钥（AIK）是一种由 SRK 生成的私有密钥，且有各种用途，包括平台认证。

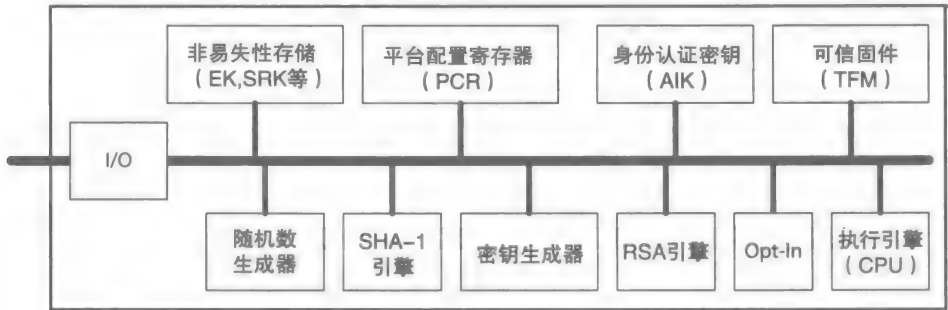


图 11-28 TPM 结构 [58]

508

随机数生成器可以根据需要由芯片中的热噪声 [59] 产生。RSA 引擎用于 RSA 的加密和解密。RSA 密钥生成器用于生成对称 RSA 密钥。SHA-1 引擎用于哈希功能。

Opt-In 模块允许用户根据平台厂商的私有指导选择 in 或者选择 out。如果 opt-in 机制有效，用户在使用功能或者服务之前会收到提示。opt-in 机制默认无效。如果 opt-out 机制有效，用户在保留或禁用功能或者服务时会收到提示，opt-out 默认有效。

11.11 安全处理器体系结构

SP 可以实现多核安全执行模式（SXM）对于给定 TSM 来创建目标安全执行环境，正如 11.4 节中讨论的。SP 可以让软件工程师选择程序的安全等级，如仅完整性、仅保密性，或者两者都有，且不管保护机制是否用于程序代码（指令和静态数据）、程序动态数据，或者程序代码和数据 [30]。尽管程序的静态数据不会改变，动态数据在程序执行中生成，包括由及时编译器动态生成的代码。

11.11.1 程序代码完整性

在代码完整性安全执行模式（CI-SXM）下执行的程序（例如可信软件模块、TSM）可以不受欺骗和拼接攻击。重放攻击在这里不作讨论，因为程序代码（包括静态数据）在程序执行中不会改变。哈希值用于认证在执行过程中指令和静态数据的正确性。然而，因为现代处理器芯片包含高速缓存和块中的缓存事物，每块（高速缓存线）中的哈希值是足够的。

在内存中程序代码块的哈希值有两种组织方法 [31, 55]。一种方法是在每一块指令和静

态程序块中嵌入哈希值，如图 11-29 所示。例如，假设 SP 是 32 位 RISC 处理器且最低（例如 L2）cache 线都为 64B，图中展示了嵌入于每一块中的 128 位（16B）密钥哈希（例如 HMAC）。命名为代码块的每 48B 程序代码生成一个哈希值，其可以包含 12 个 4B 指令或者静态数据字。一块包含 64B 内存空间。

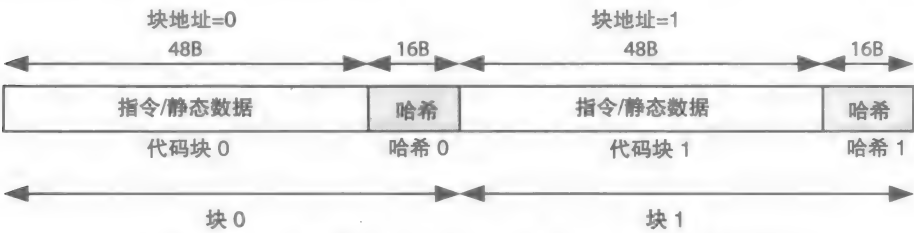


图 11-29 嵌入哈希值的程序块，假设每块大小为 64B[55]

加载进 cache 中的每个代码块必须通过嵌入哈希的每一块哈希计算和比较来认证。如果两个哈希值匹配，则块被认为是合法的且在块存入 cache 之前哈希字节变为 NOP（无操作）指令。另一方面，如果两个哈希值不匹配，则块将会在 cache 中被标记为非法的，且会抛出异常来终止程序的运行。

此外，哈希值也可以被分离存储不嵌入在块中。一种方法是建立 72 位 ECC（错误校正码）同步动态随机存储器（SDRAM）模块。在这种情况下，72 位内存空间由 64 位程序代码和在 8 位 ECC[31] 中的 8 位哈希值组成。然而 SP 需要使用拥有 128B cache 线的最低级 cache。二进制程序将被分为 128B 大小的块，每块有 16 个大小为 64 位的页。每 128B 大小的块产生一个 128 位（或 16B）密钥哈希值，且在每块预留的 16 ECC 域中存储为 16 个 8 位哈希值，如图 11-30 所示。当 cache 未命中，16 个 72 位内存页从内存传输至 SP 中。每一个 72 位内存页包含 64 位程序代码和 16 个 8 位哈希值中的一个。如果 SP 计算生成的 16B 哈希值与内存中读取的 16B 哈希值匹配，cache 线将被标记为合法的。16B 哈希值不会存储在 cache 中。虽然 16B 哈希值不能用于错误校验，但是它可以用于检测多位错误。



图 11-30 用 ECC SDRAM 设计的内存中程序指令和静态数据块；哈希值存储在为 ECC 位预留的空间中

程序编译

之前我们讨论过对于代码完整性的两种不同的内存块结构。考虑如图 11-29 所示的块结构，其哈希值嵌入在每一块中。在这种情况下，每个内存块只包含部分指令或者数据。这样在计算跳转 / 分支地址时，编译器需要考虑哈希字节的位置。

11.11.2 运行安全机制

SXM 程序的二进制代码不仅需要安全地发布和安装，在执行过程中也需要安全地装载进内存中。安装和装载程序必须遵循参考文献 [55, 60, 61] 中概述的过程类似的一组流程。在接下来的小节中，我们将讨论一系列对于软件发布和安装的安全机制，例如装载位二进制代码到内存中执行。

1. 二进制代码安全发布

如果开发 TSM 用于公共发布，二进制纯文本程序代码 (binary_{ptxt}) 需要计算其哈希值以防程序中发生无授权变更。二进制文件的哈希值用私有密钥 (PR_{company}) 来产生头记录。记录将会被附于二进制文件中以安全传送。公共发布的示例将会在稍后举例，用 E_{asym} 来表示对称密码 (如 RSA)，H 表示 (标准) 哈希算法，例如 SHA-256，“PU” 表示公共密钥。

```
//binary package includes both header and program binary
General distribution: {headercompany, binaryptxt};
//secret header record for code integrity
headercompany = Easym (PRcompany, {H(binaryptxt), "SHA-256"});
```

使用设备的公共密钥 (PU_{device}) 可以将二进制代码传输到特殊设备中，如下：

```
headerdevice = Easym (PUdevice, {H(binaryptxt), "SHA-256"});
Target device distribution: {headerdevice, binaryptxt};
```

对于特殊设备传送，头记录只能由使用其私有密钥 (PR_{device}) 的设备加密；程序只能在目标设备中执行。

2. 程序安全安装

对于 TSM 的安全安装，安装程序需要访问 SP 的 SRK，且可能需要由安装程序的授权方 (一个人) 提供的密码 [61]。安装程序在安装过程中在 SP 中生成特定程序的签名密钥 (例如 K_{sym-prog-sign}) 且用其来计算程序代码块的哈希值。

安全安装程序的一种方法是用 TFM 安装程序来保证安装程序免受攻击。在这种情况下，安装程序输入参数，例如 TSM 二进制安装文件的大小、在内存中的位置和在固件可以安装程序之前由操作系统存储的安装密钥信息 (例如 PU_{company})。

图 11-31 展示了用如图 11-29 所示的块组织的 CI-SXM 程序的安装步骤。每个的代码块哈希值存储在用于建立程序块的代码块中。

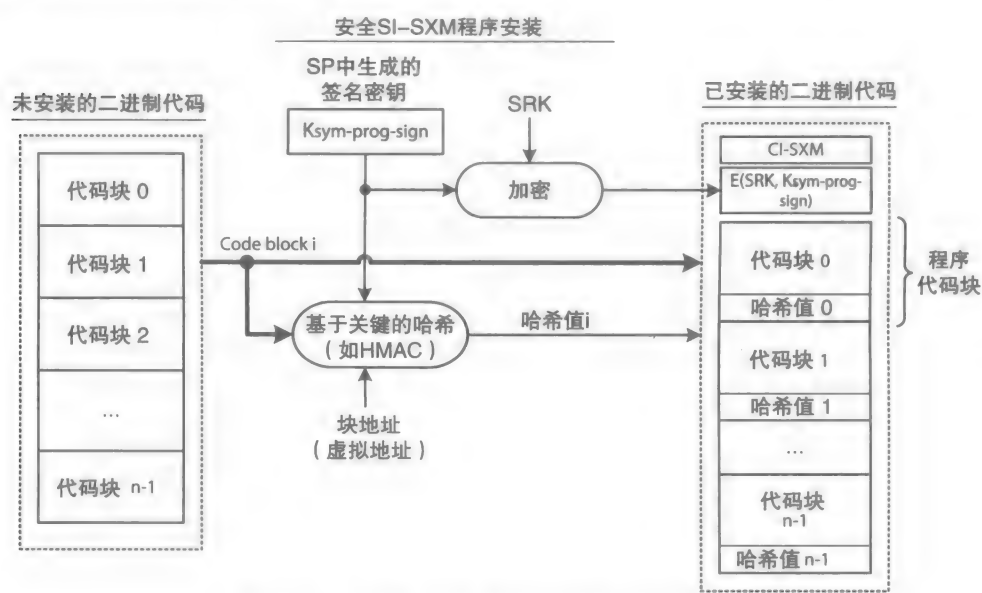


图 11-31 完整 SXM 程序代码的安全安装 [30]

在执行过程中，每个代码块的哈希值将会检测到欺骗攻击。然而，为了使得代码块哈

希值也能检测到拼接攻击，每块的起始地址，在这里也称为块地址，也用于每块哈希值的计算。在安装程序生成的签名密钥 $K_{\text{sym-prog-sign}}$ （例如随机数）用二进制文件存储（在硬盘中）之前，其也用 SRK 处理器进行加密，如图所示。以下将简单介绍如图 11-29 块组织结构 CI-SXM 的程序安装步骤。在这里， H_{keyed} 也表示密钥哈希值， code_block_j 表示程序代码（指令和静态数据）第 block_j 部分， n 为代码块数量， HV 表示哈希值。

CI-SXM 安装程序步骤：

```
blockptxt-j = {code_blockptxt-j, HVj}; //do this for all blocks,
//assume organization in Figure 11.29
//compute hash to detect spoofing and splicing attacks
HVj = Hkeyed (Ksym-prog-sign, {code_blockptxt-j, block_addressj});
binaryinstalled = {"code integrity, HMAC-128", E(SRK, Ksym-prog-sign),
{{ blockptxt-j } for j = 0 to n-1}};
```

3. 安全装载可执行二进制代码

在程序能够执行之前，为了加密安装程序生成的签名密钥 $K_{\text{sym-prog-sign}}$ 及其存储在 SP 的特殊寄存器中，装载程序必须有访问 SP 中 SRK 的权限。就像安装程序一样，基本装载程序也是 TFM。安装程序和装载程序固件都必须不泄露处理器机密。由操作系统访问和存储内存中的装载输入参数，这样在 TSM 能够启动执行之前，装载程序就可以提取签名密钥。

根据 TSM 的保护级别，装载程序可能需要在程序开始执行之前执行附加的初始化步骤，将稍后讨论这些步骤。

11.11.3 程序代码保密性

TSM 在代码（指令和静态数据）保密安全执行下的编译执行模式（CC-SXM）必须在硬盘和内存中给予保密。然而，对于指令和静态数据完整性的保护，则不需要在此模式下进行；因此也不需要哈希值计算。安装固件（参见 11.11.2 节）生成了一个加密对称密钥 $K_{\text{sym-prog-enc}}$ 来对程序代码模块进行独立加密。此外，为了阻止信息泄露，块地址需要包含在块加密中，以防两个块拥有相同的内容。下列将展示如何用块地址来为对称密钥密码（ E_{sym} ）建立一个 IV，如 AES。

为 CC-SXM 安装程序步骤：

```
IVj = {block_addressj, 00...0}; //IV is padded with 0 to make up,
//for example, a 128-bit IV for 128-bit AES
blockctxt-j = Esym (Ksym-prog-enc, IVj, blockptxt-j); //encrypt each block
binaryinstalled = {"code confidentiality, AES-128", E(SRK, Ksym-prog-enc),
{{blockctxt-j } for j = 0 to n-1}};
```

11.11.4 程序代码的完整性和保密性

CICC-SXM 这种安全执行模式，实现了对 TSM 代码（指令和静态数据）的完整性和保密性的保护。这就是先前讨论的 CI-SXM 和 CC-SXM 的组合。

为 CICC-SXM 安装程序的步骤：

```
IVj = {block_addressj, 00...0}; //create a unique IV for each block
//The following two steps may be combined using AES-CBC-MAC
code_blockctxt-j = Esym (Ksym-prog-enc, IVj, code_blockptxt-j); //encrypt
//code block
HVj = Hkeyed (Ksym-prog-sign, {code_blockctxt-j, block_addressj}); //hash
//each code block
```

```
blockctxt-j = {code_blockctxt-j, HVj} ; //combine to create
//blocks as in Figure
//11.29
binaryinstalled = {"code integrity and confidentiality,
AES-CBC-MAC", E(SRK, Ksym-prog-enc), E(SRK,
Ksym-prog-sign), {{blockctxt-j} for j = 0 to n-1}};
```

11.11.5 程序数据完整性

动态数据在程序执行过程中生成，且相对于静态数据，动态数据在内存中的值会改变。在数据完整安全执行模式（DI-SXM）下编译执行的 TSM 必须防止受到重放攻击和欺骗、拼接攻击。此前讨论过，重放攻击可以用一个旧值代替甚至更新在内存中的值。就像在 CI-SXM 中讨论的一样，单独为每一块计算的哈希值将不能检测到重放攻击。哈希树需要对数据块检测重放攻击。图 11-32 展示了数据块的二叉哈希树，叶结点表示数据块，父结点表示哈希块。假设每块大小都为 32B 且每个父结点块都有两个 128 位从其子结点计算得出的哈希值。

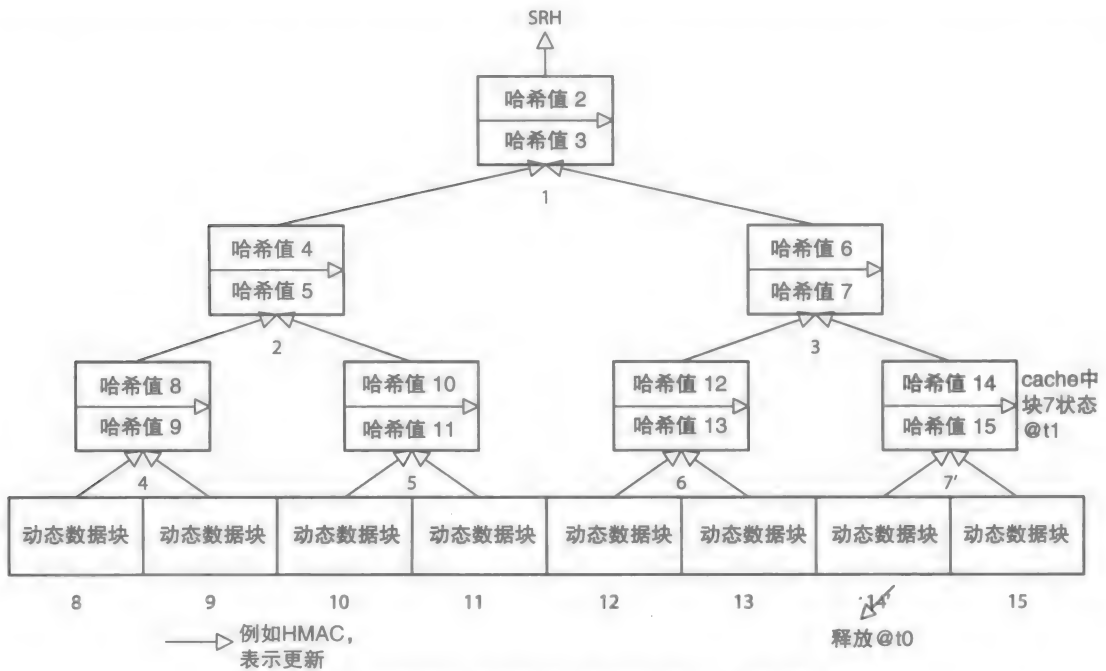


图 11-32 动态数据块中更新哈希树过程展示；数字为块地址

在这种情况下，装载固件（参见 11.11.2 节）生成一个会话签名密钥（例如 $K_{\text{sym-session-sign}}$ ）且在 TSM 用 DI-SXM 模式执行之前用此密钥来创建一个初始哈希树。图中也阐释了内存中树的结构。在每块下的数字为块地址。叶结点块在内存中的高地址部分，而父结点块在低地址部分。块 1 的哈希值（根块）被作为安全根哈希值（SRH）存储在 SP 中；注意块 0 没有被用到。

514

理论上说，每一次被修改的块被最低级的 cache 释放且离开 SP 安全边界的时候，一个新的 SRH 将会被计算。这就要求在从被释放的块到根块路径中的父结点块更新，如图 11-26 所示。然而，在实践中，因为 cache 的内存在 SP 中且被认为是安全的，一旦在从叶节点块到 SRH 路径上的父结点块在 cache 中被找到，父结点块的更新就可以停止。

例如，假设叶结点块 14 和其父结点块 17 都在 cache 中，块 14 被更新，在图中标记为

14'。现在假设块在时间点 t_0 ，块 14' 从 cache 中释放（用向下箭头表示）。因为块 7 存储在 cache 中且合法，存储在块 7 中的块 14（哈希值 14）将会在时间点 t_1 被替换为新计算出的哈希值 14'。这里不需要继续更新父结点块 3、块 1 和 SRH。下一次块 14' 从内存中取出并在 cache 中装载时，假设块 7' 依旧在 cache 中，则其值依旧为哈希 14'，且为最新的哈希值 14'。

现在，假设块 15 从内存中装载进来且块 7' 仍然在 cache 中。哈希值 15 必须与存储在块 7' 中的哈希值比较。因为哈希值 15 最初是由块 15 中的数据计算而得，还一直保存在块 7' 中，如果块 15 的哈希值与哈希值 15 相匹配，则认为块 15 是合法的。同理，块 7' 的父节点块块 3，当块 7' 被从 cache 中释放时进行更新；块 1 在块 3' 从 cache 中释放时进行更新；且 SRH 在块 1' 从 cache 中释放时进行更新。这样的处理减少了维持哈希树的开销，这将在 11.12 节中详细讨论。

如果动态数据空间在程序中被事先声明且其空间被提前装载至物理空间中，图 11-32 中的树结构就会建立起来。另一方面，为了在程序执行过程中动态分配内存空间，会用到一种不同的页机制（也可以参见第 9 章）。在此处，哈希树是一棵虚拟树且其结点是虚拟地址空间的块结构。组建此哈希树的一种方法是建立只含有根结点页和叶结点页的两层哈希树。每一页中包含若干块。例如，4KB 大小的页可以包含编号为 1 ~ 64 的 64B 大小的块。这样的哈希树可以用以下步骤建立 [30]：

- 1) 在每个叶结点页中用嵌入哈希值的方法组织动态数据块，如图 11-29 所示。例如，假设每个哈希值都为 16B，每 64B 动态数据块包含 48B 数据和 16B 哈希值。一个 4KB 的叶结点页包含 3072B ($48B * 64$) 动态数据和 1024B ($16B * 64$) 哈希值。

- 2) 计算每个叶结点页所有嵌入哈希值的校验和（用按位异或逻辑）并将其存入根结点页的块中。4KB 根结点页最多可以存储 256 个在 64 块中的 16B 校验和——每块中有 4 个 16B 哈希值。此外，根结点页可以保存 1 ~ 256 叶结点页的校验和。

- 3) 计算所有在根结点页中的校验和的累计校验和（再次用按位异或逻辑）并将它们作为 SRH 存入 SP 中。

515

如有需要，更多的叶结点和根结点页将会动态地加载。虚拟哈希树也能保护整合回硬盘中的叶结点页和根结点页。如果硬盘中的页发生了一个未授权的改变，下次当被改变的页重新存储至内存中且此页中的块被 SP 访问时，这个改变将会被检测到。

因为每一个装载进 cache 中的数据块完整性必须被确认，安全装载固件（之前讨论过）必须为这些预装载来执行程序的数据块建立一个初始哈希树。

11.11.6 程序数据保密性

DC-SXM 和 CC-SXM 类似，除了由于在程序执行过程中在内存中的数据块发生的改变会被攻击者找出，例如，内存中的数据块在不同的时间会有相同的值。为了预防这样的信息泄露，我们需要对数据块进行随机化加密 [33]。每当 cache 中改变的数据块被驱逐，除了块地址，唯一的由装载器生成的会话加密密钥 ($K_{sym-session-enc}$) 也被用来加密数据块。装载器固件也用于在 TSM 程序执行之前对任何装载后的数据块进行初始的随机加密。与其他密钥相似，会话密钥在 SP 中安全保存。

在这种情况下，即使随着时间变化块内容保持不变，但是块的加密备份仍然会改变。而且，因为动态数据是在执行过程中生成的，大多数分配给每一数据块的唯一数字存储在内存中且下一次将会访问从内存装载的加密块。对于随机加密，有两种生成唯一数据块数字的方法：

1) 随机序列。数据块的随机数序列是随机生成的。下面展示了一用随机序列对数据块 $data_block_j$ 随机加密的过程。 RN_j 代表被分配给 $data_block_j$ 的随机数, Y_j 表示用于存储 RN_j 的内存位置。密码的 IV(如 AES) 是用块地址和分配给其唯一随机序列生成的。 $IV = \{block_address, RN\}$ 可能用 0 来填充去生成加密 IV 正确的大小。每次 cache 未命中 $data_block_j$ 时, 其当前的 RN, 存储在 Y_j 位置中, 被读取出来生成 IV 来对块加密。每当块(不管改变与否)被 cache 丢弃, 将会生成新的 RN。因此, 在装载器进行初始化过程中并没有必要对每块都用到初始的 RN(例如 RN_0)。然而, 用初始 RN 将会化简处理器的结构。

```
//Initialization by loader;  $RN_0_j$  is saved at memory
//location  $Y_j$ 
 $data\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, RN_0_j\},$ 
 $data\_block_{ptxt-j}\});$ 

//1st time evicted from cache, save  $RN_1_j$  at location  $Y_j$ 
 $data\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, RN_1_j\},$ 
 $data\_block_{ptxt-j}\});$ 

//2nd time evicted from cache, save  $RN_2_j$  at location  $Y_j$ 
 $data\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, RN_2_j\},$ 
 $data\_block_{ptxt-j}\});$ 

//3rd time evicted from cache, save  $RN_3_j$  at location  $Y_j$ 
 $data\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, RN_3_j\},$ 
 $data\_block_{ptxt-j}\});$ 

etc.
```

必须保证分配给每块的随机数是唯一的; 但是事先我们没有办法来保证这一点, 所以在某些情况下, 对于某些块来说随机数并不是唯一的。例如, 用 32 位的短整数来表示随机数, 如果给定的块被访问很多次, 那么生成重复的随机数的几率就会增高。另一方面, 尽管对每块用大随机数或许可以降低重复的概率, 但是会用到更多的内存空间来存储大随机数。

2) 中序序列。一系列的不重复的数字是为每块按序列生成的, 例如 0、1、2 等。下列展示了用中序序列对 $data_block_j$ 随机加密的过程, 假设初始中序值为 0, 由装载器分配且存储于 Y_j 的内存位置中。

```
//Initialization by loader; 0 is assigned and saved at
//location  $Y_j$ 
 $block\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, 0\},$ 
 $data\_block_{ptxt-j}\});$ 

//1st time evicted from cache;  $Y_j + 1 = 1$ ; 1 is saved at
//location  $Y_j$ 
 $block\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, 1\},$ 
 $data\_block_{ptxt-j}\});$ 

//2nd time evicted from cache;  $Y_j + 1 = 2$ ; 2 is saved at
//location  $Y_j$ 
 $block\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, 2\},$ 
 $data\_block_{ptxt-j}\});$ 

//3rd time evicted from cache;  $Y_j + 1 = 3$ ; 3 is saved at
//location  $Y_j$ 
 $block\_block_{ctxt-j} = E(K_{sym-session-enc}, \{block\_address_j, 3\},$ 
 $data\_block_{ptxt-j}\});$ 

etc.
```

尽管用中序序列可以保证当块被 cache 丢弃时分配给每块的数值是唯一的, 但我们必须保证在程序执行过程中每个序列不会被用尽。例如, 如果内存中一个动态数据项每 100ns 更新一次, 那么在 $429 (2^{32} * 100ns / 10^9ns)$ 秒或大约 7.16 分钟内 32 位的中序序列值(例如 0, 2,

3, ..., $2^{32} - 1$) 会溢出。如果用 64 位中序序列, 则会每 58.5K 年溢出一次。然而, 与大随机数相似, 大中序序列数序列也会占用更多的内存空间。

通常有两个方法来缩小中序序列所占的存储空间:

1) 用短中序序列。在此情况下, 每次在程序执行过程中, 一块的中序序列值发生溢出时, SP 将停止程序的执行, 在程序可以重新执行之前生成一个新的会话密钥 ($K_{\text{sym-session-enc}}$) 且用新密钥和在序列中的初始唯一值 (例如 0) 对所有数据块进行加密 [30, 33]。然而, 如果这种机制用于操作数据块大数的 TSM 上, 对所有数据块重新加密所需的时间可能会非常长。

2) 用切片中序序列。在这种情况下, 块将组织成小组, 例如, 每组有 256 块。16 位中序唯一数值被分配给每块且一个更长的 (例如 48 位) 中序唯一数值将会在一组数据块中共享 [62]。为了实现小组中每一块的随机加密, 小组共享的 48 位唯一数值和块中 16 位私有的唯一数值串联起来形成了块的 64 位中序唯一数值。然而, 每当小组中的短序列溢出时, 对应的共享 48 位唯一数值将会增加, 小组中的私有短中序序列将会重新初始化, 且小组中所有的块将会重新加密。因为不需要新的会话密钥且小组中的块数目是相对于 TSM 中的数据块数目来说比较少, 用切片中序序列, 对于小组中的块, 用于重新加密的总时间比第一种方法要少。以下代码中, 用 $\text{data_block}_{i,j}$ 来表示小组 i 中的数据块 j 。每次当 $\text{data_block}_{i,j}$ 被 cache 丢弃, 其被分配的 16 位私有唯一数值就会增加, 当经历了 65 536 次丢弃之后, 块共享的 48 位唯一数值会增加且将 16 位私有唯一数值重新初始化 (例如 0) 来对组 i 中的所有 (256) 块进行加密。假设装载器存储了初始的 48 位唯一数值 (例如 0) 分配给组 i , 存储于内存中的 X_i 位置中且初始短私有 16 位唯一数值 (例如 0) 分配给 $\text{data_block}_{i,j}$ 存储在 Y_j 中, 下列展示了用切片中序序列对 $\text{data_block}_{i,j}$ 加密的过程:

```
//Initialization by the loader;  $X_i = 0$ ;  $Y_j = 0$ .
block_blockctxt-i,j = E( $K_{\text{sym-session-enc}}$ , {block_addressj, 0,
0}, data_blockptxt-i,j)
//1st time evicted from cache;  $X_i = 0$ ;  $Y_i + 1 = 1$ ; save 1
//at  $Y_j$ 
block_blockctxt-i,j = E( $K_{\text{sym-session-enc}}$ , {block_addressj, 0,
1}, data_blockptxt-i,j)
//2nd time evicted from cache;  $X_i = 0$ ;  $Y_i + 1 = 2$ ; save 2
//at  $Y_j$ 
block_blockctxt-i,j = E( $K_{\text{sym-session-enc}}$ , {block_addressj, 0,
2}, data_blockptxt-i,j)
//3rd time evicted from cache;  $X_i = 0$ ;  $Y_i + 1 = 3$ ; save 3
//at  $Y_j$ 
block_blockctxt-i,j = E( $K_{\text{sym-session-enc}}$ , {block_addressj, 0,
3}, data_blockptxt-i,j)
. . .
//65535th time evicted from cache;  $X_i = 0$ ;  $Y_i + 1 = 65535$ ;
//save 65535 at  $Y_j$ 
block_blockctxt-i,j = E( $K_{\text{sym-session-enc}}$ , {block_addressj, 0,
65535}, data_blockptxt-i,j)
//65536th time evicted from cache;  $X_i + 1 = 1$ ;  $Y_i = 0$ ; 1
//is saved at  $X_i$ ; 0 is saved at  $Y_j$ 
//A firmware in SP encrypts all the blocks in group  $i$  and
//then resumes program execution
for(j = 0; j < number_of_blocks_in_group_i; j++)
    block_blockctxt-i,j = E( $K_{\text{sym-session-enc}}$ , {block_addressj,
    1, 0}, data_blockptxt-i,j);
```

518

分配给最相关块的唯一数值或许会被存储在 SP 中一个特殊的 cache 中, 以提高性能。

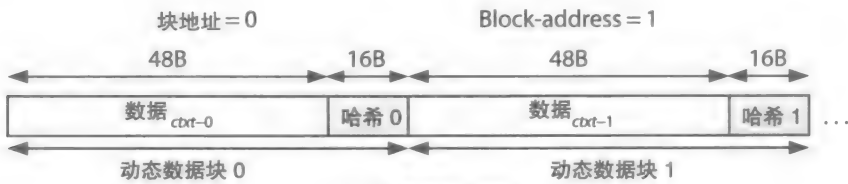
所有（方法 1）或者某些（方法 2）块的重新加密任务由 TSM 处理器的部分进行管理。如果 TSM 中断了，其对应的重新加密任务将会停止，且将会在 TSM 重新启动时重新执行。

11.11.7 程序数据的完整性和保密性

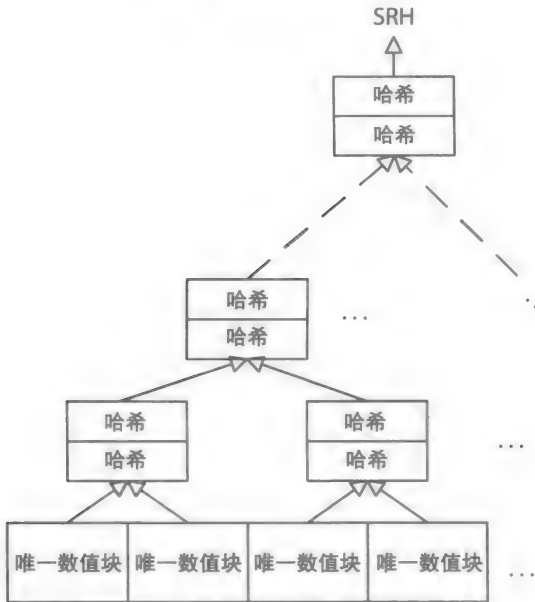
在 DIDC-SXM 中，每个动态数据块必须被加密，就像在 DC-SXM 中一样，且必须被哈希且必须包括一棵哈希树，就像在 DC-SXM 中一样。此外为了保护每一个数据块的完整性，每块必须被分配给唯一的数值，用来实现随机加密过程，这个过程也需被保护。然而，实验证明不是每一个数据块都需要维护一棵哈希树，且另一棵哈希树可以被分配给一个唯一数值来检测重放攻击 [63, 64]。唯一数值单独的哈希树，比数据块的哈希树要小一些，对于检测重放攻击更有效。此外，唯一数值不需要加密 [65]。数据块仍然被哈希，如下列展示，但是不需要为了数据块维护一棵如图 11-32 中的哈希树。每个数据块的哈希值或许可以被嵌入到每块中或者单独存储，如图 11-29 和图 11-30 中的代码块所示。

图 11-33 展示了有嵌入哈希值的动态数据块的组织结构和数据块分配唯一数值的对应哈希树。在图中，每块假设为 64B 且包含一个 48B 的数据块且 16B（128 位）的哈希值。假设每个动态数据块有长（64 位）唯一数值，8 个这样的数值可以存储在每个 64B cache 块中（8 = 64B/64bit）。

519



a) 动态数据块组织



——> 哈希 密钥

b) 动态数据为唯一数值的哈希树

图 11-33 保护动态数据：a) 嵌入哈希值的动态数据块；b) 动态数据块唯一数值的哈希树

图 11-34 展示了用如图 11-33 所示的数据块组织和分配给每块的任意唯一数值的复杂重放攻击。攻击重放了 cache 块（数据和嵌入哈希）和分配给它的唯一数值。攻击在时间 t2 用在时间 t1 生成的旧副本替换了块 1 和分配给它的唯一数值 23。然而，因为唯一数值的哈希树将会检测到 17 是一个非法的数值，执行过程将会停止，防止攻击实现其目标。

11.11.8 程序代码和数据的完整性及保密性

CICC-SXM 和 DIDC-SXM 组合提供了最大的程序保护。这个组合执行模式需要 4 个密钥，分别是由安装器固件生成的 $K_{\text{sym-prog-enc}}$ 和 $K_{\text{sym-prog-sign}}$ 、由装载器固件生成的 $K_{\text{sym-session-enc}}$ 和 $K_{\text{sym-session-sign}}$ （11.11.2 节）。前面两个密钥用于对程序代码块（包括静态数据）加密和哈希。后两个密钥用于对程序数据块的加密和哈希、对分配用于数据块动态加密唯一数值的数据块的哈希树进行维护。这就需要两种块必须可分辨以便于 SP 在执行过程中可以对不同类型的块使用不同的密钥。以下是两种可能的解决方案：

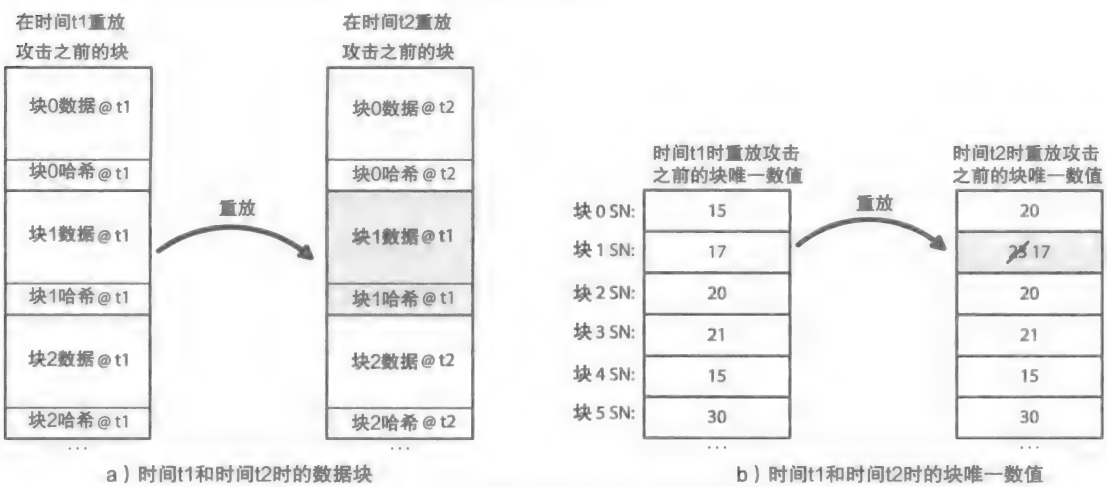


图 11-34 块的重放攻击和其序列数值的展示

1) 物理内存结构。一种选择是将主（物理）存储空间进行切片成两个区域：一个没有 SXM，包括 DMA 访问区域和一个 SXM 区域，还包含一棵哈希树区域。然而这个选择需要一个安全的内核（操作系统的一部分）或者一个由受信任程序生成的 DMA 转换。

如果为代码和数据预留 SXM 内存空间被分割成代码和数据区域，这样如果地址代表代码区域（第 8 章图 8-5），SP 会用程序密钥来解密且验证传入的 cache 块。另一方面，如果内存地址代表一个数据块，SP 会用会话密钥来解密且验证传入的 cache 块。

另一方面，如果 SXM 虚拟页映射到预留的 SXM 内存空间的任意位置（哈希树区域之外）且用到了额外的 SXM 数据块的随机加密，则数字 0 可能会被分配给每一个代码块且非 0 唯一数值分配给每一个数据块 [33]。每次当最底层 cache 缺失时，如果块唯一数值为 0，则这个块就被认为是一个代码块，且程序密钥将会用与解密和验证块；而如果唯一数值非 0，则代表一个数据块，SP 则用会话密钥来解密和验证块。如之前提到的，0 和非 0 唯一数值不需要解密。然而，哈希树会用于验证 0 和非 0 的分配数值。

2) 虚拟内存结构。另外一个选择是用额外的虚拟内存空间，分成 SXM 和非 SXM 的虚拟空间来存储哈希树。用分开的虚拟空间可以使得 SXM 程序（例如 TSM）可以使用一整个

SXM 虚拟地址空间分为代码和数据区域。假设物理地址被 cache 命中 (第 10 章), 被 cache 命中的任意哈希树块, 其虚拟块地址也被存储在最低层 cache 中, 这样就可以用来决定其对应父哈希节点的虚拟地址。此外, 分开的向前翻译缓冲区 (TLB) 可以将哈希树虚拟页数值翻译为其对应的物理页数值。为提高效率, 哈希树块可以存储于分离 cache 内存中。如果虚拟内存地址 (存储在 cache 中) 表示一个代码块, 则 SP 可以用程序密钥来对进入的 cache 块进行解密和验证。另一方面, 如果虚拟内存地址表示的是一个数据块, SP 可以使用会话密钥来对进入的 cache 块进行解密和验证。此外, 如果数据块被随机加密过了, 存放在虚拟内存空间为哈希树及其叶节点块预留的空间中的分配给每一个数据块的唯一数值不需要被加密。

11.11.9 处理中断

中断要求在产生中断时 CPU 的状态 (例如, 寄存器内容和中断返回地址) 被记录下来, 且在终端程序重新控制时被重新存储起来 (第 9 章)。在 SXM 中, 寄存器内容和返回地址必须被安全地存储以检测攻击。SP 中需要的额外资源的总数取决于 SP 是否在同一时间只用于执行一个 SXM 程序 (例如单 TSM 处理器)[55] 或者同时执行多个 TSM (SXM 多处理器)[31-33]。

1. 单安全执行环境

在这种情况下, 任意时间只有一个 TSM 在 SXM 中执行, 表示为 SXM-OP (单处理器)。这样只有一个 SXM 状态需要在中断过程中被保护。这个过程可以由用 SP 的 SRK 来对 SXM 寄存器内容进行加密和哈希实现。此外, 寄存器的加密过程可以是随机的, 由此来防止用唯一数值进行加密的寄存器信息泄露 (例如随机生成的随机数)。加密的寄存器内容随后被存储至它们各自的寄存器中以便中断处理程序 (IH) 可以将它们存储至内存中。哈希值、中断返回地址和随机数 (如果存在) 将会安全地存储在 SP 中。在这种过程中需要的密钥和它们的密钥材料, 还有 SRH (如果存在) 也会在 SP 中保存。

在 SP 中, 寄存器和 cache 线被标记出来——例如, SXM 的 1 和非 SXM 的 0。任意对非 SXM 处理器中标记 1 的寄存器或标记 1 的 cache 线读 / 写或者任意对 SXM 处理器中的标记 0 的寄存器或标记 0 的 cache 线读 / 写都可以产生异常。SXM 处理器可以通过将其标记改为 1 对任意寄存器和数据块进行写操作。中断产生时, SP 清除所有 SXM 标记的寄存器、刷新所有 SXM 标记的 cache 数据块, 且将 SP 从 SXM 改变为非 SXM 并将控制权交回给 IH。

在时间共享环境中, SP 除了可以执行 TSM (SXM 程序) 也可以执行非 SXM 程序。当 TSM 在运行时, OS 不能开始执行另一个 SXM 进程。当从一个中断返回时, SP 比较返回地址和原先内部存储的地址。如果两个地址相匹配, SP 就换成 SXM, 对重新存储加密寄存器的内容进行解密, 并且重新启动 SXM 进程。反之, 返回地址如果表示一个非 SXM, 则代表需要重新启动对非 SXM 进程的中断 (11.11.8 节), 或者表示一个攻击, 这样会抛出异常。

2. 多安全执行环境

在这种情况下, SP 用于在时间共享环境中执行多 SXM 和非 SXM 进程。密钥表用于存储中断 SXM 进程状态下的进程 ID。例如, 非 0 ID 用于标记 $ID = 0$ 的 SXM 进程及一个非 SXM 进程。SP 资源 (例如寄存器和 cache 线) 用进程 ID 标记。这个表可以是一个 SP 的私有密钥表 (嵌入) 也可以是一个存储在 SP 外部的虚拟密钥表。

如果使用私有密钥表, 则 SP 只能在时间共享环境中执行 SXM 固定的数值的进程。如果发生了中断, 密钥、返回地址、寄存器内容等就会存储在私有表中, SXM 寄存器将会清空; SXM 数据块会在控制权交回给 IH 前清空。相反的, 除了将寄存器内容存在私有表中,

我们可以维护一个更大的表，寄存器内容可以在 SXM-OP 中处理。寄存器内容将被加密、哈希并存储回内存中的 IH 寄存器中，且哈希和其他信息存储在表中。

另一方面，因为虚拟密钥表可以在硬盘中复制，其大小可以根据需要增长以允许在时间共享环境中同时执行任意数目的 SXM 进程。然而，处理器状态（密钥、寄存器内容、返回地址等）在进程 ID 下的虚拟表中存储之前需要用 SP SRK 进行加密。虚拟密钥表页可以映射至物理页中（第 10 章）。与 11.11.5 节中类似的哈希树用于验证与虚拟密钥表相关的内存页面。

与有在片上私有密钥表相比，维护一个虚拟密钥表需要增加对中断处理延迟。然而如果与虚拟密钥表相关的最近访问物理页地址存储在 SP 中特殊的 cache 内存中并能快速访问，这个延迟可以减少。

当用私有或者虚拟密钥表时，从中断中重新启动类似于对 SXM-OP 的描述。然而，由于同一时间可能有不止一个 SXM 进程在运行，返回地址将会与存储在进程 ID 下的密钥表进行比较；如果两个地址匹配，SXM 进程将会重新启动执行。

11.12 设计示例：安全处理器

本节展示了包括数据通路和示例 SXM 指令的 SP 架构。数据通路包含了一个标准的处理器核心和安全执行所需的模块。也将会展示一个简单的 TSM 应用。

11.12.1 SP 特征

下面的列表列出了示例 SP 的特征的缺陷：

- 1) 在任意时间 SP 只支持一个 SXM 进程执行（例如 SXM-OP，参见 11.11.9 节）。523
- 2) SP 能实现 CICC-SXM 和 DIDC-SXM 的组合以实现对程序最大的保护。在本节中，SXM 项表示包含保密性和完整性可信软件模块（TSM）的代码和数据的最大程序保护。
- 3) SP 包含了可以启用或禁用 SXM 的 SXM 指令集。
- 4) SP 包含了表示一个 SP 为 SXM 或者非 SXM 的 SXM 的状态位集合。
- 5) TSM 代码块（包括静态数据）被加密和哈希来检测代码欺骗和拼接攻击（如果存在）。代码块由图 11-29 模式的编译器构成。在需要的时候，“程序块”项用于关联一个嵌入在代码块中的 cache 块和哈希值，如图所示。此外，TSM 项和“SXM 程序”将互换使用。“SXM 进程”表示运行的 TSM。
- 6) 执行过程中会动态改变的 SXM 程序数据块将会被加密且哈希树会用于检测动态数据欺骗、拼接和重放攻击（如果存在）。
- 7) 动态数据块的加密过程不是随机的。
- 8) SXM 有自己完整独立的链接库以便在编译时进行静态链接。程序不需要依赖外部库或者系统进程。
- 9) 在编译过程中 SXM 静态地声明数据块且分配内存空间；在执行过程中并不会分配内存空间。
- 10) 主（物理）存储器中的一个区域将会为 SXM 预留出来。这个区域分为程序代码（包括静态程序数据）和动态数据区域。最主要的地址位定义了每一个区域；0 表示代码区域且 1 表示数据区域。数据区域又分为数据和哈希树区域，如图 11-35a 所示。有 4 个叶结点（动态）块的动态数据块的哈希树展示于图 11-35b 中。
- 11) SXM 程序很小以至于可以适合其整体的 SXM 代码区域。这样，在 SXM 中不会有

虚拟到物理地址转换的过程。

12) L2 是最低级的 cache 内存且使用如 MESI 协议的回写一致性协议 (第 10 章)。

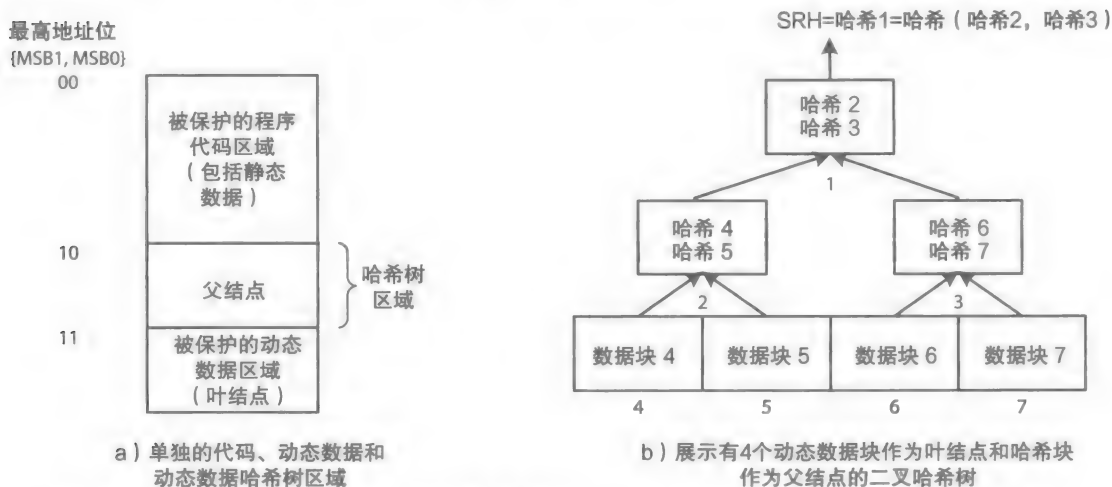


图 11-35 内存中的 SXM 程序代码和数据组织: a) 内存映射; b) 哈希树 (用二叉树展示)

13) SP 包含一个在硬件中实现的加密 / 解密和哈希引擎 (EDHE) 作为一个 SP 中的嵌入式系统。其用于对进入的 SXM 代码块解密和哈希, 对进入的 SXM 动态数据块解密, 且对出去的更改过的 SXM 数据块加密。

14) SP 同样也包含在硬件中实现的哈希树引擎 (HTE), 也是一个在 SP 中的嵌入式系统。它被用于验证进入的用哈希树的 SXM 数据块且当更改过的数据块被 L2 cache 丢弃时更新哈希树。

15) EDHE 和 HTE 都需要由主板制造商安全安装的可信固件。

16) SP 的周长是系统的安全边界。这样 cache 是安全的且可以包含纯文本指令和数据。

17) SP 也可以包含可信程序安装器和装载器固件 (11.11.2 节)。装载器固件与 OS 通信来完成 3 种任务:

a. 装载器固件提取两个由安装器生成的程序密钥 $K_{\text{sym-prog-enc}}$ 和 $K_{\text{sym-prog-sign}}$ 并将它们存储在 SP 中。

b. 装载器生成两个会话密钥 $K_{\text{sym-session-enc}}$ 和 $K_{\text{sym-session-sign}}$ 并将它们存储在 SP 中。会话密钥在每一次 SXM 程序开始执行的时候都会改变。

c. 装载固件建立最初 TSM 数据块的哈希树。数据块初始内容可以是未知的。

一旦装载固件完成这些任务, OS 就开始执行 SXM 程序。

18) 中断的处理方法类似于 SXP-OP。

19) 为了简化, 我们假设 SP 核心包括用于处理中断的硬件。在这里, 我们将关注 EDHE 和 HTE 上的数据通路。

11.12.2 处理器架构

图 11-36 展示了 SP 的数据通路。其包括一个处理器核心, L1 和 L2 cache, 还有实现 SXM 所需的模块。寄存器集用于存储密钥, 包括由安装器固件生成的 $K_{\text{sym-prog-enc}}$ 和 $K_{\text{sym-prog-sign}}$, 还有其他由装载器固件生成的密钥, 和两个由装载器固件生成的用于在执行过程中保护

TSM（动态）数据块的会话密钥 $K_{\text{sym-session-enc}}$ 和 $K_{\text{sym-session-sign}}$ 。

EDHE 负责对进入的 SXM 程序块或者 SXM 数据块进行解密和哈希，且对更改过的出去数据块进行加密。SXM 程序块包含一个代码块和一个嵌入哈希。代码块包含指令和 / 或静态数据。因为代码块的内容一般不会改动，所以这些块在被替换的时候将被 cache 删除。HTE 负责对 SXM 数据块的哈希树进行维护。EDHE 和 HTE 将稍后在本章中讨论。

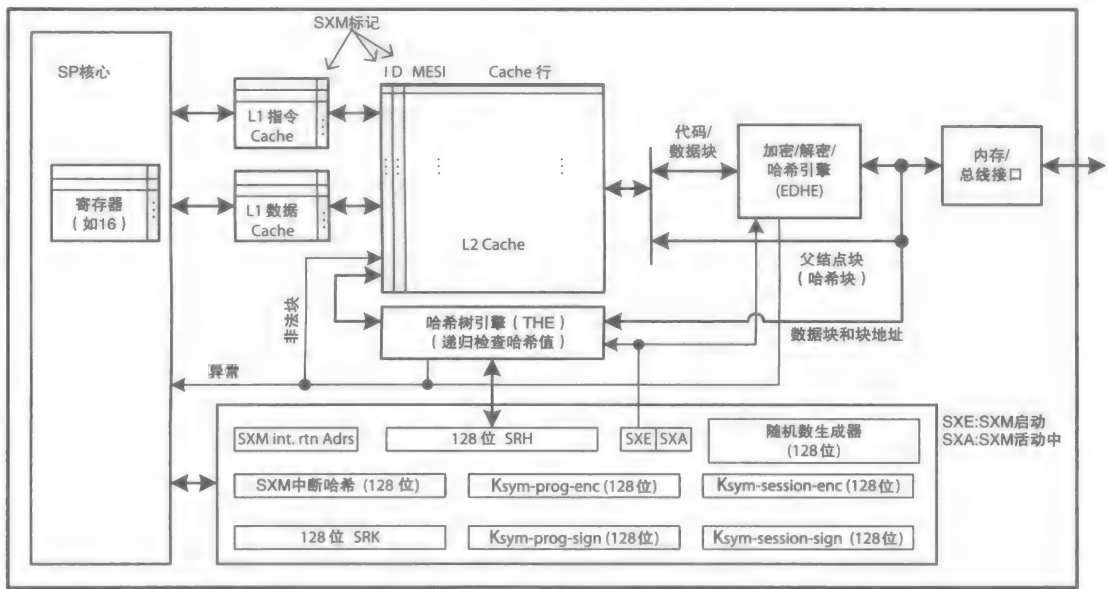


图 11-36 示例 SP 的数据通路。SXM 实现了 CICC 和 DIDC 安全执行模式

SP 数据通路还包括一个 2 位 SXM 状态寄存器 [55]；这两个寄存器位被称为安全执行使能标识 (SXEF) 和安全执行活动标识 (SXAF)。SXEF 可以对 EDHE 和 HTE 进行启动。SXAF 用于保证当前只有一个 SXM 进程在运行。当 SXAF 信号有效，只要当前有一个 SXM 正在运行，它将会阻止 OS 启动另一个 SXM 进程。SP 可以是表 11-10 中三种合法模式的其中一个。SXM 处理器的中断会重置 SXEF，将其标记为 0，且从 SXM 中断设置 SXEF 返回 1。

526

表 11-10 基于 SXM 状态值位 SZAF 和 SXEF 的 SP 状态

| SXAF | SXEF | SP 状态 |
|------|------|--|
| 0 | 0 | SP 在非 SXM 状态且当前执行一个非 SXM 进程。OS 可以交换一个 SXM 进程 |
| 0 | 1 | 非法状态（未用到） |
| 1 | 0 | SXM 进程被中断且没有在运行。当前执行程序是一个非 SXP 进程。OS 不可以替换另一个 SXM 进程；否则，将会抛出异常 |
| 1 | 1 | SP 在 SXM 状态中且当前执行一个 SXM 进程 |

如图 11-36 所示，当前进程的寄存器内容和 cache 块被标记为 1 (SXM) 或 0 (非 SXM)。一个 SXM 进程可以只读 SXM 标记寄存器内容和 cache 块。然而进程可以对任意的寄存器或者数据块进行写操作，将标记改为 1 (SXM)。另一方面，非 SXM 进程可以对非 SXM 标记寄存器内容或者非 SXM 标记的 cache 块进行读或者写操作。在组合 L2 Cache 中的所有块也被标记为非 SXM 或 SXM。此外，在 L2 中的指令块被标记为 “I” 且数据块被

标记为“D”。这样就防止 SXM 进程将指令块当作数据块进行访问。

表 11-11 展示了 SXM 指令集。其他参考书中也有相似指令 [31, 33, 35]。

表 11-11 SXM 指令集

| 指 令 | 描 述 |
|-------------|--|
| SXM_ENTER | 如果 SXAF = 0，这个指令将 SXAF 和 SXEF 都置为 1（活动的）从而让 SP 进入 SXM 状态。SXEF = 1 启动了 EDHE 和 HTE。两个由装载器固件生成的会话密钥 $K_{\text{sym-session-cnc}}$ 和 $K_{\text{sym-session-sign}}$ 将用于对 SXM 数据块进行加密/解密和哈希。如果 SXAF 为 1 将会抛出异常 |
| SXM_EXIT | 如果 SXAF = 1 且 SXEF = 1，SP 退出 SXM 状态。该指令将清除 SXAF 和 SXEF 将它们设置为 0，重置 SXM 标识寄存器、刷新 L1 数据 cache，且将 L2 cache 中的 SXM 标记数据块进行废止。如果退出是临时的，L1 指令 cache 和 L2 中的指令块将不会被刷新 |
| SXM_LD | 如果 SXAF = 1 且 SXEF = 1，指令将从 SXM 动态数据块进行装载。如果指令的执行引起了 cache 缺失，从主存中接收的块将在装载至 L2 cache 之前被解密且被标记为 SXM。HTE 唤起来验证 SXM 数据块。如果块没有被验证，将会抛出异常来中止 SXM 进程。如果 SXEF 为 0 时也会抛出异常 |
| SXM_ST | 如果 SXAF = 1 且 SXEF = 1，指令安全地将数据从 cache 存至内存中。特别地，指令将对 cache 中的块进行写和更新操作且将其标记为 SXM。如果指令引起了 cache 缺失，块将首先装载进 cache 中，标记为 SXM，然后再进行更新。HTE 将被唤起来验证进入的 SXM 数据块。cache 中块的 SXM 标记与块中所有字进行关联；SP 不会处理部分标记块。如果 SXEF 为 0 将会抛出异常 |
| LD_FROM_SXM | 如果 SXAF = 1 且 SXEF = 1，指令将 SXM 数据块装载至寄存器中且将寄存器标记为非 SXM，允许寄存器内容被非 SXM 指令使用。如果指令引起了 cache 缺失，在被装载进 L2 cache 之前，块将会被加密。HTE 被唤起来验证进入的 SXM 数据块。如果块未被验证，将会抛出异常来中止 SXM 进程。如果 SXEF 为 0 也会抛出异常 |
| ST_TO_SXM | 如果 SXAF = 1 且 SXEF = 1，指令将非 SXM 寄存器内容存储至 SXM 数据块中。如果指令引起了 cache 缺失，在装载至 L2 cache 之前块将会被加密且标记为 SXM。HTE 被唤起来验证进入的 SXM 数据块。如果块未被验证，将会抛出异常来中止 SXM 进程。如果 SXEF 为 0 也会抛出异常 |

应用示例：安全加密服务

考虑一个实现加密 API 的 TSM。应用软件和商业操作系统，会用 API 来对应用或者操作系统级数据进行加密。例如，考虑一个用 API 且在密钥链中指定一个密钥数值，其纯文本数据在内存中的内存起始地址和目的密码文本在内存中的内存起始地址（参见 11.8.2 节中的示例）。以下列出了 TSM 对应用数据进行安全加密的步骤：

- 1) TSM 执行指令“SXM-ENTER”。如果 SXAF 为 0，就将 SP 换成 SXM。否则如果 SXAF = 1，则应用必须等到当前执行（不同的）SXM 进程中断，重置 SXAF（将其标记为 0），且将控制权返回 OS，然后才可以开始安全加密 TSM 进程。
- 2) 当 TSM 被唤醒时，它用“LD”（第 8 章）和“ST_TO_SXM”指令将应用纯文本、非 SXM 数据复制到主存中的 SXM 数据区域中。
- 3) TSM 将从应用提供的密钥链中安全地提取出加密密钥。
- 4) 用“SXM_LD”和“SXM_ST”指令，TSM 安全地对纯文本进行加密，然后存储至内存中的 SXM 数据区域、密钥文本也将存储至 SXM 数据区域。
- 5) 最后，使用“LD_FROM_SXM”和“ST”（第 8 章）指令，TSM 将生成的密钥文本

才能过 SXM 数据区域复制到内存中的应用密钥文本（非 SXM）数据区域中。

11.12.3 加密解密哈希引擎

在 SP 中，EDHE 包含了加密、解密和哈希函数，作为嵌入式系统实现。当 SXEF 被设为 1 时其可以运行多项任务。EDHE 解密、哈希和验证进入的程序 cache 块，它们都包含了代码块和嵌入哈希。在执行“SXM_LD”“SXM_ST”“LD_FROM_SXM”或者“ST_TO_SXM”指令时，EDHE 可以对进入的数据块进行解密，且可以对更改的有 SXM 标记的数据块进行加密。注意到 SXM 数据块的随机加密是在示例 SP 中实现的。未改变的标记为 SXM 的数据块将在块被替换时被 cache 删除。SXM 程序或数据块的块地址也用于块加密、解密和哈希。哈希中的块地址包容用于检测拼接攻击（如果存在）。

527
528

cache 线验证：代码块

地址位中最重要的两位（MSB1 和 MSB0）定义了两种不同的块；(00)₂ 定义为程序块且 (11)₂ 定义为数据块（图 11-35a）。 $K_{\text{sym-prog-enc}}$ 用于对每一个 SXM 程序块进行解密，且 $K_{\text{sym-prog-sign}}$ 用于对每一个 SXM 代码块进行哈希。会话加密密钥 $K_{\text{sym-session-enc}}$ 用于对 SXM 数据块进行解密 / 加密。

对于进入的 SXM 程序块（代码块加嵌入哈希），如果代码块计算的哈希与嵌入哈希匹配，程序块中的哈希值将在程序块被存至 L2 cache 中之前被 NOP 指令代替。此块将在 cache 中置为合法的且标记为 1（SXM）。否则，cache 线将会置为非法的（I）且会抛出异常，这样将会中止 SXM 进程（阻止攻击）。

如果进入的 SXM 数据块是执行“SXM_LD”或者“LD_FROM_SXM”指令的结果，块将被认为是一个 SXM 数据块。此块将从内存中装载且在其存至 L2 cache 之前进行加密（用 $K_{\text{sym-session-enc}}$ ）。数据块和其块地址也被装载进 HTE 进行验证。然而，cache 中的块被认为是合法的且程序将会继续正常运行除非 HTE 抛出异常，标记一个攻击。由于写缺失，对于进入 SXM 数据块的处理不管是用“SXM_ST”还是“ST_TO_SXM”结果是一样的——块被解密且装载进 cache 中，HTE 被唤醒来验证块，且块在 L1 数据 cache 中被更新且在 L1 数据 cache 和 L2 cache 中被标记为已更改。

图 11-37 展示了 EDHE 和 HTE 在读循环中的数据通路，且图 11-38 展示了写循环的数据通路。

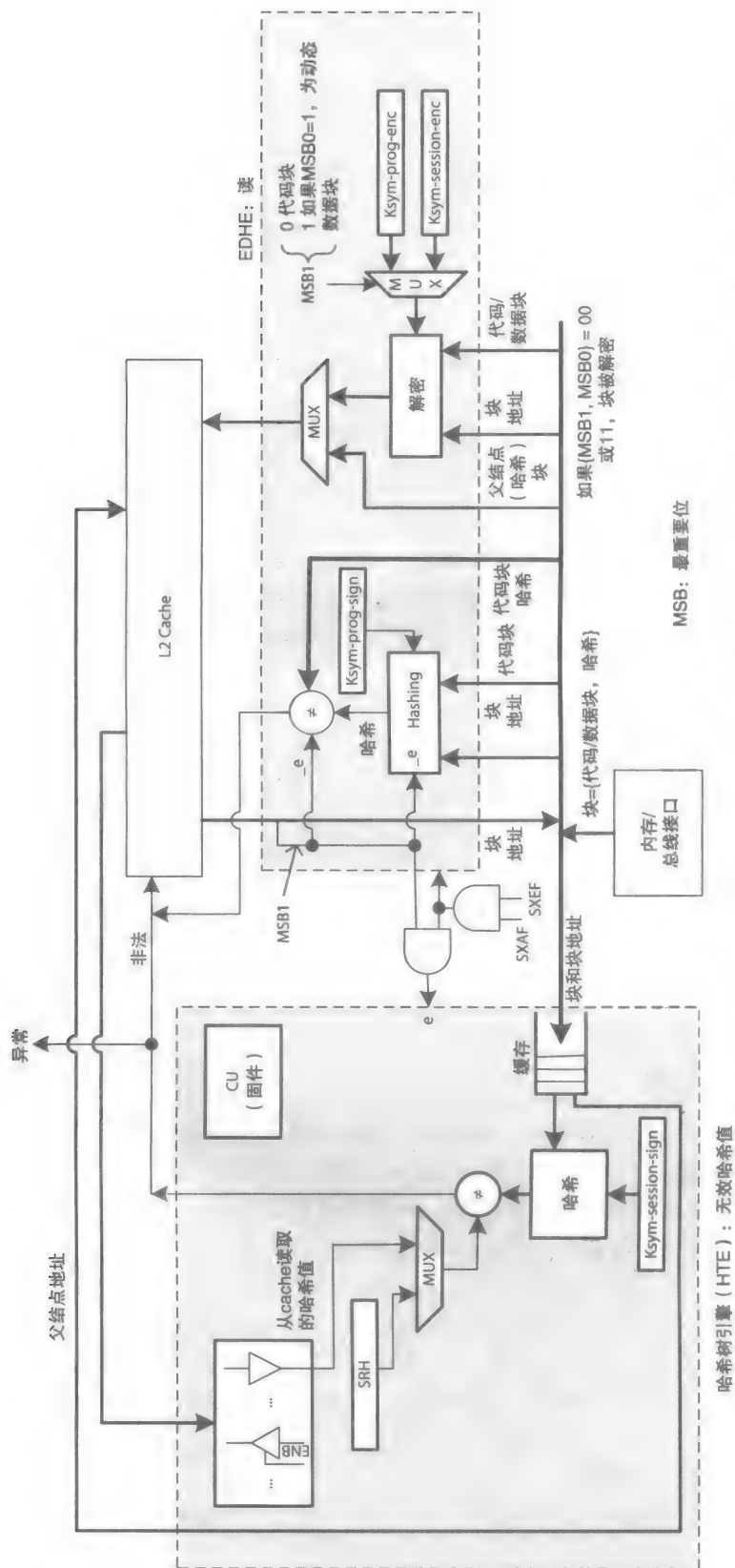
11.12.4 哈希树引擎

HTE 负责对哈希树进行解析，从而对进入的 SXM 数据块进行验证，且当更改的 SXM 数据块被 L2 cache 丢弃且从 SP 中的安全边界离开时，对哈希树进行更新。然而，因为树的解析需要时间，验证过程是在后台运行的；不需要数据推测执行。在这种情况下，数据推测执行意味着处理器继续执行程序，但不会将计算结果提交给寄存器，除非推测执行中的数据已被验证过。然而，因为除了重新启动程序以外，目前没有比较普遍的办法来从攻击中恢复过来 [66]，所以 SP 可以把计算结果提交给寄存器。在 cache 缺失的时候，装载进 L2 cache 的 SXM 数据块被标记为合法的（例如，在 MESI 协议中的 E 或 S 状态），且程序还是照常继续执行直到 HTE 抛出异常，表示检测到攻击。异常会引起 SP 中止进程且把控制权交还给 OS；程序仍然可以重新运行。

529

1. cache 线验证：数据块

HTE 验证数据块的数据通路也展示在图 11-37 中。HTE 递归地在哈希树中的结点上操作，用图 11-35b 的示例哈希树，如表 11-12 所示。



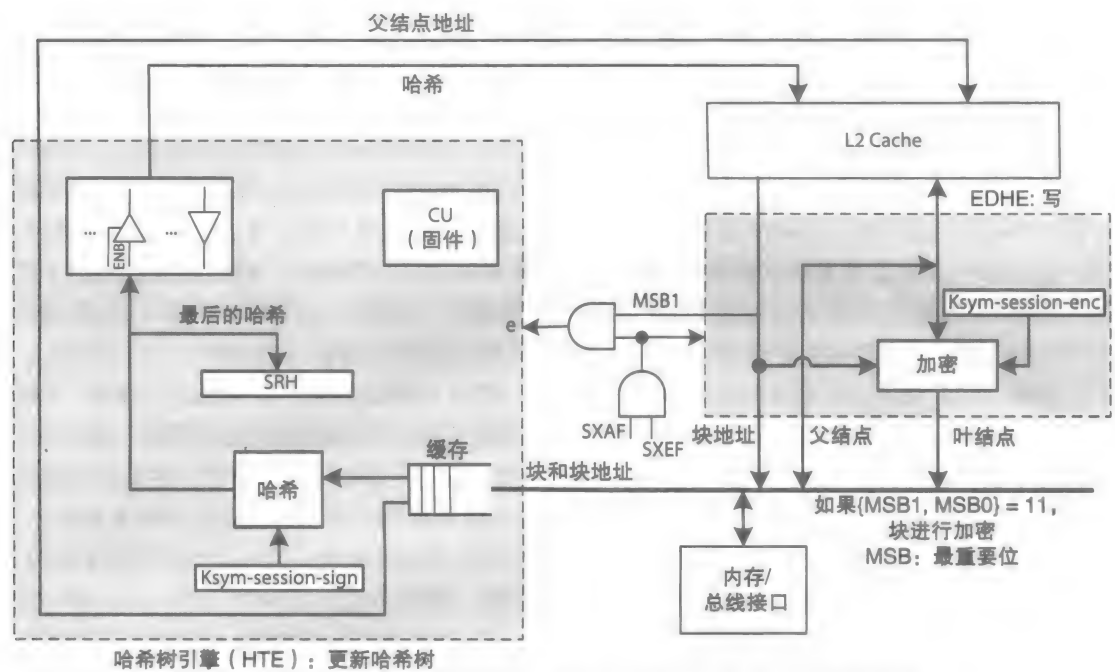


图 11-38 回写数据块至内存；丢弃一个更改过数据块的数据通路

表 11-12 用哈希树进行动态数据块验证实例

| 读取的数据块 | 缓存中的块 | 目标父结点 | Cache 动作 | 动作 | Cache 中的块 |
|--------|-------|-------|----------|----------------------------|---------------|
| 进入 5 | 5：验证 | 2 | 读缺失 2 | 需要 Block2, 进入 2 | 5 |
| | 2：验证 | 1 | 读缺失 1 | 需要 Block1, 进入 1 | 5, 2 |
| | 1：验证 | SRH | 读命中 1 | SRH=? Hash1, 移除 1 | 5, 2, 1 |
| | 2：验证 | 1 | 读命中 2 | Block1.Hash2=? Hash2, 移除 2 | 5, 2, 1 |
| | 5：验证 | 2 | 读命中 2 | Block2.Hash5=? Hash5, 移除 5 | 5, 2, 1 |
| 进入 4 | 4：验证 | 2 | 读命中 2 | Block2.Hash4=? Hash4, 移除 4 | 5, 2, 1 |
| 进入 7 | 7：验证 | 3 | 读缺失 3 | 需要 Block3, 进入 3 | 5, 2, 1, 7 |
| | 3：验证 | 1 | 读命中 1 | Block1.Hash3=? Hash3, 移除 3 | 5, 2, 1, 7, 3 |
| | 7：验证 | 3 | 读命中 3 | Block3.Hash7=? Hash7, 移除 7 | 5, 2, 1, 7, 3 |

“进入”：块与其块地址进入 HTE 的缓存中

“移除”：块与其块地址从缓存中移除

假设 cache 最初是空的且 SP 访问的第一个数据项在 SXM 数据块 5 中。当内存找到块 5，则块 5 被解密且被 EDHE 存至 L2 cache 中。在这里，此块将被复制到 L1 数据 cache 中。此时，在所有 cache 中块 5 的状态都是合法的。当块被装载进 SP 时，块 5 和其块地址也被存至 HTE 的缓存中。随后 HTE 验证块 5。根据块地址，HTE 确定块 2 是块 5 的父结点块且会尝试从 L2 cache 进入块 2 中。然而，因为 cache 初始为空，块 2 会引起一个缺失。当内存提供块 2，块被存储至 L2 cache 中且与其块地址一起存至 HTE 中。注意到父结点块包含了哈希值且在装载进 L2 cache 前就不需要解密步骤，如图 11-37 所示。

此时，HTE 尝试访问块 1，根结点块也是块 2 的父结点块。再次地，块 1 不在 cache 中引起了一个缺失。其从内存读出且装载至 L2 中，与其块地址进入 HTE 的缓存中。所有块，

530
531

包括哈希块在装载时都被标记为合法。因为块 1 是根结点块，所以 HTE 计算且用其哈希值 (Hash1) 与存储在 SP 中的 SRH 比较。如果 Hash1 与 SRH 匹配，块 1 就被认为验证通过且在缓存中清除，然后在缓存中的块 2 将会被进行验证。

此时，HTE 计算块 2 的哈希值 (Hash2) 且将其与块 1.Hash2 比较。如果两个哈希值匹配，块 2 验证通过且从缓存中移除，还剩下块 5 在缓存中等待验证。最后，计算块 5 的哈希值 (Hash5) 且与块 2.Hash5 比较。如果两个哈希值匹配，块 5 验证通过且从缓存中移除。这个过程将中止哈希树对块 5 的验证解析。如果在树解析任意时间内，任意两个哈希值不匹配，则 HTE 会抛出异常，在中止 SXM 进程且把控制权交回给 OS 之前，这会引入 SP 清除所有 SXM 标记的寄存器且刷新 L1 数据 cache 和所有 SXM 标记的 L2 数据块。

如表 11-12 所示，内存访问的第一个块是块 5，HTE 需要花费好几步来验证。然而，验证其他的块如块 4，只需要一步就可以完成验证。这是因为块 2 作为块 4 的父结点块已经存在于 L2 cache 中 (假设没有被替换) 且在验证块 5 时已经验证过，如表所示。所以，在验证其他块时，由于在验证块 5 时已经验证过，从块 2 到 SRH 的数据通路上的哈希值都不需要重新验证。

在最后的例子里，SXM 数据块 7 装载进 cache 中且与其块地址一起进入 HTE 的缓存中。其哈希值 (Hash7) 需要计算并且与块 3.Hash7 比较。然而，块 3 此时并不在 cache 中，块 3 从内存中装载至 L2 cache 中且与其块地址进入 HTE 的缓存中。因为块 1 已经验证过了且在 cache 中合法，如表所示，Hash3 被计算且与块 1.Hash3 比较。如果两个哈希值匹配，块 3 验证通过且从缓存中移除，留下块 7 进行下一步验证。HTE 计算 Hash7 且与块 3.Hash7 比较。如果两个哈希值匹配，数据块 7 被认为验证通过且从缓存中移除。

注意，当在验证块 5 时，块 4 和块 7 可能会进入缓存中。HTE 根据先进先服务 (FCFS) 的顺序验证数据块。

2. 哈希树更新

在 cache 丢弃一个更改过的 SXM 数据块时，HTE 也会重新计算一个新的 SRH。在表 11-13 中展示了用图 11-38 中的数据通路和图 11-35b 中的哈希树的示例过程。

表 11-13 哈希树更新示例

| 被丢弃的数据块 | HTE 缓存中的数据块 | 目标父结点 | Cache 控制操作 | 动 作 | Cache 中的块 |
|---------|-------------|-------|------------|--------------------------------|------------|
| 进入 5' | 5': 更新 2 | 2 | 写缺失 2 | 需要 Block2, 进入 2 | |
| | 2: 验证 | 1 | 读缺失 1 | 需要 Block1, 进入 1 | 2 |
| | 1: 验证 | SRH | | SRH=? Hash1, 移除 1 | 2, 1 |
| | 2: 验证 | 1 | 读命中 1 | Block1.Hash2=? Hash2, 移除 2 | 2, 1 |
| 进入 4' | 5': 更新 2 | 2 | 写命中 2 | Block2.Hash5 ← Hash5', 移除 5' | 2', 1 |
| | 4': 更新 2 | 2' | 写命中 2 | Block2.Hash4 ← Hash4', 移除 4' | 2'', 1 |
| 进入 7' | 7': 更新 3 | 3 | 读缺失 3 | 需要 Block3, push 3 | 2'', 1 |
| | 3: 验证 | 1 | 读命中 1 | Block1.Hash3=? Hash3, 移除 3 | 2'', 1, 3 |
| | 7': 更新 3 | 3 | 写命中 3 | Block3.Hash7 ← Hash7', 移除 7' | 2'', 1, 3' |
| 进入 2'' | 2'': 更新 1 | 1 | 写命中 1 | Block1.Hash2 ← Hash2'', 移除 2'' | 3', 1' |
| 进入 1' | 1': 更新 SRH | SRH | | SRH ← Hash1', 移除 1' | 3' |
| 进入 3' | 3': 更新 1' | 1' | 写缺失 1' | 需要 Block1, 进入 1' | 3' |
| | 1': 验证 | SRH' | | SRH'=? Hash1', 移除 1' | 3', 1' |
| | 3': 更新 1' | 1' | 写命中 1' | Block1'.Hash3 ← Hash3', 移除 3' | 1'' |

' 和 " 表示更改的次数，' (一次) 且 " (两次)；“进入”块与其块地址进入 HTE 缓存中；“移除”从缓存中移除

假设 SXM 数据块 5' 在 L2 cache 中, 这里 ' 表示块被更改过。假设块 5' 被 cache 丢弃, 将会被加密且复制到内存中。为了对哈希树进行更新操作, 加密过的块 5' 与其块地址在离开 L2 cache 的同时将被复制到 HTE 的缓存中。因为 SP 中的 cache 被认为是安全的, HTE 只需要更新块 2, 即块 5 的父结点块。假设块 2 不在 cache 中, 这将引起 cache 缺失。块 2 将从内存复制到 L2 cache 中且被 HTE 进行验证, 则块 1 也需要被验证, 如之前讨论过的循环一样。

L2 cache 中的块 2' 内容将变为 {Hash4, Hash5'}, {} 表示串联, 且块 5' 从缓存中被移除, 完成哈希树的更新。接下来, 假设块 4' 被 L2 cache 丢弃。假设块 2' 还在 cache 中, 这个更新将不会引起 cache 缺失且会非常快地将块 2" 变为 {Hash4', Hash5'}, " 表示两次更新。块 7' 的丢弃需要块 3 更新 Hash7'。然而, 假设块 3 在 cache 中缺失, 则块 3 将从内存中读出, 验证然后更新。这样会引起块 3.Hash7 在 cache 中被 Hash7' 替换且块 7' 会从缓存中移除。

接下来, 假设块包含 {Hash4', Hash5'} 的块 2" 被 cache 丢弃。块 2" 与其块地址将进入 HTE 的缓存中。根据其地址, HTE 确定块 1 为块 2 的父结点块, 必须要更新。假设块 1 仍然在 cache 中, 经过更新后, 块 1' 内容变为 {Hash2", Hash3}, 且块 2" 从缓存中移除。

表 11-13 同样也展示了块 1' 被丢弃的过程, 这将引起 HTE 去更新 SRH 中的 Hash1', 且展示了块 3' 的丢弃过程, 引起块 1' 重新加载且验证, 然后更改 Hash3'。注意到, 当离开 SP 时哈希块不需要被加密, 且加载至 cache 时不需要被解密。

相比于标准处理器, SP 引入了额外的开销。如图 11-37 所示, 每一个 SXM 程序和数据块在装载进 L2 cache 之前都需要解密, 需要密钥。此外, 哈希树解析或者更新都产生了额外的 cache 流量, 这可能减缓下一个 SXM 进程的执行。然而, 我们可以使用单独的 cache 来存储哈希块以增强效率。

11.13 延伸阅读

我们除了提供了一些介绍和背景知识, 也讨论通过硬件来保护密钥链, 内存验证机制和通过建立 SXM 来划分执行环境。下列是一些其他运行硬件检查类型的示例:

- 处理器或许可以实现一种用来监测缓冲区溢出攻击 [67, 68] 的在硬件中的安全返回地址栈 (SRAS)。尽管恶意软件可以引起缓存溢出和用一个新的地址 (例如病毒地址) 对内存栈进行欺骗, 返回地址将与存在处理器中 SRAS 中的地址不一样, 这样就不会使得地址跳转去执行病毒。这个机制与用 11.11.9 节中的私有或者虚拟表来限制返回地址 (并不是所有类型的进程) 很像。
- 硬件阵列边界检查器使用基本地址和阵列的大小来监控溢出边界错误 [69]。
- 硬件监控器可以监测不正常的程序行为。这些包括在硬件中建立区域内和进程间的控制流量监控器 [70]。监控器可以是一个基于 FSM 的检查器也可以是在编译期间动态监控跳转 (区域内) 和调用 / 返回 (进程间) 地址的程序控制流和数据流图像。FSM 和表都可以用于跟踪所有允许的呼叫 - 被叫关系。表中存储了调用 / 返回地址且将地址映射至 FSM 状态中。非法的调用 / 返回地址表示了一个引起 FSM 进入非法状态的非法行为。
- 基于分析的程序检查器核实程序是否遵循了一条正常的执行路径 [71]。所有可能的

程序路径在某些训练运行的时候都会被记录下来,且检查器利用这些记录来监测非法路径。训练时间必须足够长以便于减少判断出错的次数。

- 在硬件中动态跟踪程序信息流 [72-75]。一个完整的策略是在处理器中实现防止高完整性数据去使用 OS 标记的低完整性数据的功能。如果在进入系统时,一项输入被标记为低完整性的,例如,通过如 USB 主机控制接口的设备控制接口 (DCI) 的输入。污点数据值将被用作指令或者内存地址 (指针) 来防止被污染。门级信息流跟踪 [74] 硬件需要对每个逻辑门都设置阴影逻辑来跟踪每一位数据的可信度。每一位输入和每一位输出都被标记为可信 (0) 或者不可信 (1)。这样,简单地用一位不可信位不能总是表示结果是不可信的。这个结果还依赖于用于处理不可信位的逻辑门。例如,用一位可信输入 $x = 0$ 和一位不可信输出 $y = 0$ 或 1 的与门,结果将为 0,是可信的。用这种机制进行跟踪的指令集架构 (ISA) 数据通路要求程序指针 (PP) 不能无条件修改且不能有间接的内存加载 / 存储指令。如果条件是可信的,则 PP 的内容也是不可信的,且这样会导致所有寄存器甚至所有的内存空间的内容也是不可信的。这样所有依赖于条件的指令都必须转换为断言指令,且所有无边界的循环都必须转换为有边界的循环 (为了防止时间信息泄漏),这些有边界的循环使用有中止条件的计数器,在这个条件下,循环中的所有指令都判断为这个条件的否定。计数器由一个特殊的指令 (“counterjump”) 进行初始化且在每次迭代中减 1 直至到 0。在这种机制下,循环 (包括嵌套循环) 将作为一个实体来执行,当其完成时,将会引起 PP 增 1 且无分支退出循环。在这里我们不考虑硬件中的不可信信息流木马和可能篡改内存内容的物理攻击。可以证明,当阴影逻辑可能增加电路的规模时 (在某一研究中能增长 70%), 它不产生负面影响的时钟频率。
- 代码和数据进行替换,通过比较运行程序的多份备份的行为来检测攻击 [76]。在每次替换中,不同的内存层用来检测内存访问错误,每次复制产生不同的哈希结构来保护严格的数据完整性,且用每次复制不同的加密结构来更好地保护数据的保密性。
- 通过实现更好的内存带宽分配结构来保护可用性。存储器控制器 (MC) 通常运用多种算法来优化和安排由内存提供的优秀 cache 缺失机制。例如先到先服务 (FCFS) 机制也可以给从当前活动行 (第 7 章) 访问的列分配最高的优先级,以此来增加内存吞吐量,且下一个优先级分配给剩下的未完成请求中最旧的非活动行 [77, 78]。然而,刷新有随机 (地址) 事务 MC 的恶意线程可能会增加行访问的次数且这样将会增加其他线程的等待时间。失速时间公平内存调度 (STFM) 用内存放缓值来更好地处理内存请求 [79]。在这种情况下,存储器控制器为每一个有一个优秀事务列表线程计算一个内存放缓 (S) 值,如果有其他的线程 (T_{shared}) 共享内存,这个值将作为平均失速值的速率,如果只有这一个线程 (T_{alone}),这个值将作为期望失速值。由此, $S = T_{\text{shared}}/T_{\text{alone}}$ 。不公平参数 (U) 将被用式子 $U = S_{\text{max}}/S_{\text{min}}$, 这里 S_{max} 和 S_{min} 是所有请求中内存放缓的最大和最小值。如果 U 小于一些可接受的值 (例如 $U < a$), 则调度将用一种算法来增加内存的吞吐量——例如,通过给突发事务分配更高的优先级。另一方面,如果 $U \geq a$, 则在优秀内存请求处理中存在不公平因素, $S = S_{\text{max}}$ 的对应的线程请求将会被分配到最高的优先级。然后 FCFS 调度将用来优化所有最高优先级请求中的事务调度。

参考文献

1. M. M. Olama, J. J. Nataro, V. Protopopescu, and R. A. Coop, Security concerns and disruption potentials posed by a compromised AMI network: risks to the bulk power system, The 2012 International Conference on Security and Management (SAM'12), Las Vegas, 2012, pp. 133-137.
2. A program example illustrating buffer overflow attack, http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html.
3. Champagne David, "Scalable security architecture for trusted software," a Ph.D. dissertation, Princeton University, 2010.
4. Markus G. Kuhn, Cipher instruction search attack on the bus-encryption security micro-controller DS5002FP, *IEEE Transactions on Computers*, vol. 47, no. 10, October 1998, pp. 1153-1157.
5. Huang Andrew, *Hacking the Xbox: An Introduction to Reverse Engineering*, No Starch Press, San Francisco, 2003.
6. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber, Authentication in distributed systems: theory and practice, SOSP '91: *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 165-182.
7. Hoglund, Greg. and Butler, James, *Rootkits: Subverting the Windows Kernel*, Addison-Wesley Professional, 2005.
8. Elias Levy, Approaching zero, *Security & Privacy*, IEEE Volume: 2, Issue: 4, pp. 65-66.
9. Tal Garfinkel et al., Terra: a virtual machine-based platform for trusted computing, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003, pp. 193-206.
10. Michael Fey, Brian Kenyon, Keven Readon, Brandon Rogers, and Charles Ross, *Security Battleground: An Executive Field Manual*, Intel Press, 2012.
11. Thomas A. Fuhrman, The new old discipline of cyber security engineering, SAM '12, 2012, pp. 547-553.
12. Ruby Lee, Simha Sethumadhavan, Edward Suh, and David Grawock, Tutorial on security for computer architects, ISCA Security Tutorial, San Jose, California, June 4, 2011.
13. Adam Waksman and S. Sethumadhavan, Tamper evident microprocessors, In: *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
14. Mark S. Miller et al., *Capability Myths Demolished*, SRL, 2003, pp. 42-49.
15. Capability-Based Computer Systems, available from: <http://www.cs.washington.edu/homes/levy/capabook/Chapter1.pdf>.
16. D. E. Bell and L. J. LaPadula, *Secure Computer Systems*, Mitre Corporation, Bedford, MA, 1977.
17. K. J. Biba, *Integrity Consideration for Secure Computer Systems*, Mitre Corporation, Bedford, MA, 1977.
18. Timothy Fraser, LOMAC: Low Water-Mark Integrity Protection for COTS Environments, 2000 IEEE Symposium on Security and Privacy, 2000 (S&P 2000), pp. 230-245.
19. David D. C. Brewer and Michael J. Nash, The Chinese wall security policy. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, IEEE Press, 1989, pp. 206-214.
20. David D. Clark and David R. Wilson, A comparison of commercial and military computer security policies, *IEEE*, 1987, pp. 184-194.
21. Sally Adee, "The Hunt for the Kill Switch," *IEEE Spectrum*, May 2008, pp. 35-39.
22. K. Gandolfi et al., Electromagnetic analysis: concrete results, In: *Proceedings of 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2001, pp. 251-261.
23. D. Asonov and R. Agrawal, Keyboard acoustic emanations, In: *Proceedings of the IEEE Symposium on Security & Privacy*, May 2004, pp. 3-11.
24. Zhenghong Wang and Ruby B. Lee, A novel cache architecture with enhanced performance and security, In: *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-41)*, 2008, pp. 88-93.
25. Waksman and S. Sethumadhavan, Silencing hardware backdoors, SP '11 *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pp. 49-63.

26. Craig Gentry, A fully homomorphic encryption scheme, Ph.D. dissertation, spring 2009, Stanford University.
27. M. Hicks, S. T. King, M. M. K. Martin, and J. M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, In: *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
28. Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres, Hardware mechanisms for memory authentication: a survey of existing techniques and engines, *Trans. on Comput. Sci. IV*, LNCS 5430, 2009, pp. 1-22.
29. Champagne, David, Elbaz, Reouven, and Lee, Ruby B., The reduced address space (RAS) for application memory authentication, In *Proceedings of the 11th International Conference on Information Security* (Taipei, Taiwan, September 15-18, 2008, pp. 47-63.
30. Austin Rogers, Designing cost-effective secure processors for embedded systems: principles, challenges, and architectural solutions," a dissertation, University of Alabama in Huntsville, 2010.
31. D. Lie, C. Thekkath, M. Mitchell, et al., Architectural support for copy and tamper resistant software, *Proc. of the 9th Intl Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000, pp. 168-177.
32. G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas, AEGIS: architecture for tamper-evident and tamper-resistant processing, *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*, 2003, pp. 160-171.
33. G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas, AEGIS: architecture for tamper-evident and tamper-resistant processing, Computer Science and Artificial Intelligence Laboratory (CSAIL), MIT, 2004. (An extended version of [48]).
34. Qiong Liu, Reihaheh Safavi-Naini, and Nicholas Paul Sheppard, Digital rights management for content distribution, Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003 - Volume 21, Australian Computer Society, January 2003.
35. Search for Extraterrestrial Intelligence (SETI), <http://setiathome.ssl.berkeley.edu/>.
36. Distributed.net, http://www.distributed.net/Main_Page.
37. Auguste Kerckhoffs, La cryptographie militaire, *Journal des Sciences Militaires*, <http://www.petitcolas.net/fabien/kerckhoffs/>.
38. National Institute of Standards and Technology (NIST), <http://csrc.nist.gov/publications/>.
39. AES-NI instruction set, <http://software.intel.com/>.
40. William Stallings, *Cryptography and Network Security*, Pearson Prentice Hall, 4th ed., 2006.
41. RSA calculator, <https://www.cs.drexel.edu/~jpoppyack/IntroCS/HW/RSASheet.html>.
42. Francis Crowe, Alan Daly, and William Marnane, Scalable dual mode arithmetic unit for public key cryptosystems, *Information Technology: Coding and Computing*, Vol. 1, 2005, pp. 568-573.
43. Efficiency of ECC Cipher, <http://www.certicom.com/index.php/the-basics-of-ecc>.
44. R. Needham and M. Schroeder, Using encryption for authentication in large networked computers, *Communications of the ACM*, Volume 21 Issue 12, Dec. 1978, pp. 993-999.
45. SHA-1 calculator, <http://www.sha1.cz/>
46. Richard Spillman, *Classical and Contemporary Cryptology*, Pearson Prentice Hall, 2005.
47. Hans Brandl, *Trusted Computing: The TCG Trusted Platform Module Specification*, Infineon Technologies AG, Embedded Systems 2004.
48. Chu-Hsing Lin, Hierarchical key assignment without public-key cryptography, *Computers and Security*, Vol. 20, No. 7, 2001, pp. 612-619.
49. Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas, Silicon physical random functions, *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, 2002, pp. 148-160.
50. Yohei Hori, Hyunho Kang, Toshihiro Katashita, and Akashi Satoh, Pseudo-LFSR PUF: A compact, efficient and reliable physical unclonable function, *7th International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, Cancun, Quintana Roo, Mexico, 2011, pp. 223-228.
51. G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas, AEGIS: a single-chip secure processor, *Information Security Technical Report* (2005) 10, pp. 63-73.

52. Sundeep Bajikar, Trusted platform module (TPM) based security on notebook PCs: white paper, Intel Corporation, June 2002.
53. Jeffery Dwoskin and Ruby Lee, Hardware-rooted trust for secure key management and transient trust, CCS'-07, Alexandria, Virginia, 2007, pp. 389-400.
54. Weiping Peng, Yajian Zhou, Cong Wang, Yixian Yang, and Yuan Ping, A new hierarchical key authdata management scheme for trusted platform, *International Conference on Multimedia Information Networking and Security*, 2010, pp. 463-467.
55. Ruby Lee et al., Architecture for protecting critical secrets in microprocessors, 32nd International Symposium on Computer Architecture, 2005 (ISCA '05), pp. 2-13.
56. Ralph C. Merkle, Protocols for public key cryptography, In: *IEEE Symposium on Security and Privacy*, 1980, pp. 122-134.
57. Smart card basics, <http://www.smartcardbasics.com/>.
58. TCG Specification Architecture Overview, Specification Revision 1.2 28 April 2004, http://class.ee.iastate.edu/tyagi/cpre681/papers/TCG_1_0_Architecture_Overview.pdf.
59. Y. Wang, H. Zhang, Z. Shen, and K. Li, Thermal noise random number generator based on SHA-2 (512), in *Proceedings of the 4th International Conference on Machine Learning and Cybernetics*, Guangzhou, China, 2005, pp. 3970-3974.
60. M. Milenković, A. Milenković, and E. Jovanov, A framework for trusted instruction execution via basic block signature verification, In: *Proceedings of the 42nd Annual ACM Southeast Conference*, 2004, pp. 191-196.
61. D. Kirovski, M. Drinic, and M. Potkonjak, Enabling trusted software integrity, In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002, pp. 108-120.
62. Chenyu Yan, Rogers B, Englander D, Solihin D, and Prvulovic, M, Performance and security of memory encryption and authentication, *Computer Architecture*, 2006. ISCA '06. 33rd International Symposium on Digital Object Identifier, 2006, pp. 179-190.
63. A. Rogers, Low overhead hardware techniques for software and data integrity and confidentiality in embedded systems, master's thesis, Electrical and Computer Engineering Department, University of Alabama in Huntsville, 2007.
64. B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, Using address independent seed encryption and bonsai Merkle trees to make secure processors OS- and performance-friendly, In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, Chicago, IL, 2007, pp. 183-196.
65. Jun Yang, Lan Gao, and Youtao Zhang, Improving memory encryption performance in secure processors, *IEEE Trans on Computer*, 2005, pp. 630-640.
66. Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas, Caches and Merkle trees for efficient memory authentication, MIT-LCS-TR-857, 2002.
67. J. P. McGregor, D. K. Karig, Z. J. Shi, and R. B. Lee, A processor architecture defense against buffer overflow attacks, *Proc. IEEE Intl. Conf. on Information Technology: Research And Education (ITE 2003)*, August 2003, pp. 243-250.
68. R. B. Lee, D. K. Karig, J. P. McGregor, and Z. J. Shi, Enlisting hardware architecture to thwart malicious code injection, *Proc. Intl. Conf. on Security in Pervasive Computing (SPC-2003)*, lecture notes in computer science, Springer Verlag, March 2003.
69. Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic, HardBound: architectural support for spatial safety of the C programming language, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, 2008, pp. 103-114.
70. Divya Arora, Srivaths Ravi, Anand Raghunathan and Niraj K. Jha, Secure embedded processing through hardware-assisted run-time monitoring, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, 2005, pp. 1530-1591.
71. Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee, Anomalous path detection with hardware support, *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '05)*, 2005, pp. 43-54.
72. G. Edward Suh, Jaewook Lee, and Srinivas Devadas, Secure program execution via dynamic information flow tracking, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004, pp. 85-96.

73. Shashidhar Mysore, Bitu Mazloom, Banit Agrawal, and Timothy Sherwood, Understanding and visualizing full systems with data flow tomography, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, 2008, pp. 211-221.
74. Mohit Tiwari, Hassan M. G. Wassel Bitu, et al., Complete information flow tracking from the gates up, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIV)*, 2009, pp. 109-120.
75. Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic, FlexiTaint: a programmable accelerator for dynamic taint propagation, In: *14th International Symp. on High Performance Computer Architecture (HPCA)*, 2008, pp. 173-184.
76. Ruirui Huang, Daniel Y. Deng, and G. Edward Suh, Orthrus: efficient software integrity protection on multi-cores, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010, pp. 371-383.
77. Scott Rixner, Memory controller optimizations for web servers, *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37 2004)*, pp. 355-366.
78. Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens, Memory access scheduling, *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, 2000, pp. 128-138.
79. Onur Mutlu and Thomas Moscibroda, Stall-time fair memory access scheduling for chip multiprocessors, *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*, 2007, pp. 146-158.

练习

- 11.1 列出多种用户、组织（如军队、银行）、应用程序和系统（如个人计算机、云、手持设备）可能遇到的安全问题。
- 11.2 考虑一个政府办公室发放护照。效仿 SEM 和表 11-1 中的示例来开发一种“发放难以伪造的护照”的安全机制。
- 11.3 给出一个理由说明为什么在一个组织中强制访问控制是必要的？
- 11.4 考虑一个带有客户可以租借的保险箱的银行。银行需要选择一个方案只允许授权者访问保险箱。另外，客户希望有更多的自由，偶尔允许他们的朋友或亲戚来访问他们的箱子。为以下每个技术，逐条列记银行和业主的安全所需要做的及它们需要哪些保护，以防止未经授权的人访问一个安全性问题或拒绝一个合法人的访问。
 - a. 给出一个银行可以使用的基于 ACL 的强制性的安全方案。提示：每一位顾客给银行一系列同样可以访问这种安全机制的名字。
 - b. 给出一个银行可以使用的基于列表的功能的安全方案。提示：银行给每一位顾客 n 个密码来供给朋友或亲戚。
- 11.5 多级安全策略模型是什么？
- 11.6 多级安全策略是什么？
- 11.7 简要介绍 BLP 的 *- 属性。
- 11.8 什么政策模型可以用来防止 Stuxnet 恶意软件改变一个工业控制系统的规范？
- 11.9 Stuxnet 是为了寻找一个特定的控制系统而被称为可编程逻辑控制器（PLC）。什么政策模型可以用来防止 Stuxnet 通过网络传输控制系统信息？
- 11.10 Flame 恶意软件被设计成从计算机系统中“吸取”信息（按键、截图、音频等），然后通过因特网将它发送给控制它的人。什么政策模型可以用来防止 Flame 通过网络传输数据？
- 11.11 为什么实现了访问控制机制和系统的软件运行起来需要保持信赖？

- 11.12 假设 CPU 使用一个 8 位的超前进位加法器 (CLA)。使用两个 CLA 模块，但假如改动其中的一个使其输出错误的结果。例如，改变正确的表达式 $s_3 = p_3 \oplus c_2$ 为 $s_3 = \overline{p_3 \oplus c_2}$ 。一个典型的处理器上有数十亿晶体管，很难判断出其中有两个加法器。按如下操作：
- a. 设计一个输入触发硬件木马，在输入为 0x44 时触发输出错误的结果。
 - b. 假设你使用 50 个特定的测试向量来测试这个加法器。检测到木马的概率是多少？
 - c. 假设这个加法器是 32 位的 CLA，带有完全相同的输入触发为 0xA8888888 的木马。确定一百万次测试中能够检测到木马的概率。
- 11.13 假设一个模 24 计数器被用来创建一个定时炸弹木马。
- a. 你需要多少测试向量来触发这个木马？
 - b. 假设你使用 10M 次测试，这个电路仍然工作正常。你需要多久重置一下电源来阻止定时炸弹的触发？
 - c. 设计一个输出 1 的电路来重置可能使用计数器创建了定时炸弹木马的模块的电源。还假设测试向量的数量部分 (b) 包括后硅验证期间测试应用；也就是说，在大规模生产之前。示例芯片制造后硅验证的目的只是为了使用实际硬件代替仿真工具申请更多的测试。
- 11.14 设计以下图 11-39 所示的 16 位加密 / 解密电路。它使用一个多功能 16 位输入寄存器，一个右移寄存器作为输出寄存器和一个带有密钥的 8 位 LSFR。这个电路手工操作（不需要控制单元）。验证你的设计。

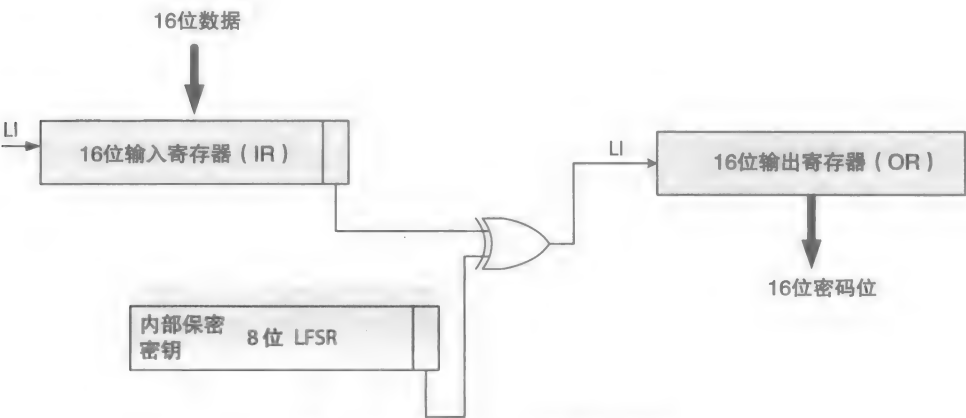


图 11-39 练习 11.14 的数据通路

- 11.15 发展的威胁的三个来源是什么？
- 11.16 除了需要更多的硬件，使用同态加密的目的是什么？将同态加密的实现变得更加困难的可能潜在问题是什么？（提示：也可以参考第 3 章中的 *FP* 算法）。
- 11.17 设计一个基于有限状态机的控制单元一次加密 / 解密 16 位的数据。图 11-40 表示了这个框图。当插入 start 信号时控制器开始工作。一个用户选了一个 16 位的输入来切换连接到 DFF 上的开关来插入 start 信号，如图中所示。一旦控制器开始工作，ack（确认）信号用来复位 DFF。这个加密 / 解密数据通路包含一个计数器来计数多功能寄存器必须移位的次数。在这个框图中，CC 表示了组合电路。当计数器达到了右边的值，操作停止，控制器返回到 standby，它的初始状态。每次加密 / 解密一个 16 位的输入时，这个系统应当仅被复位一次，工作多次。（也可以参考练习 11.14）

一次块欺骗、拼接或重放攻击，将会发生什么？每一种类型的攻击可以被测到吗？为什么？

- 11.31 假设对一个 DI-SXM 程序，数据块的序列数被存储在 SP 的内部，而不是内存中。简要说明在 SP 内部存储序列数的优点和缺点。
- 11.32 参考一个安全的虚拟内存管理系统的设计。假设我们想用二级哈希树来验证每个虚拟页面。每个哈希树包含一个根页面和多个叶子数据页面。每个叶子页面的哈希值存储在它相应的根页面中。假设页面大小为 4KB，缓冲块是 64B，每一个哈希值占 16B，需要多少根页面来验证 16MB 动态分配的虚拟内存空间？
- 11.33 参考虚拟内存数据块的一个哈希树（即这个哈希树使用虚拟地址）。假设 SP 实现物理地址缓冲。在这种情况下，一个块的虚拟地址也被保存在了低级缓冲区。简要解释为何在处理哈希树和测试重放攻击时保存虚拟地址是有必要的？
- 11.34 参考 8 个数据块。画出这个哈希树并解释通过 HTE 对块 8、9 和 14 的验证。同时，确定由于访问这些块而导致错过多少缓冲。假设没有缓冲区的父亲块被替换过。
- 11.35 我们想通过比较所需的内存空间来维持分配到内存数据块的序列数的一个哈希树。研究了两种不同的序列数：64 位与 64 位拼接（48 位的 long 和 16 位的 short）的序列数。而且，假设每 16 个连续的块用一个 long 数字，块大小是 64B，每个哈希值占 256 位，并且动态数据内存空间的最大值是 1MB。确定在每种情况中维持这个哈希树所需的内存空间大小。
- 11.36 参考一个 SXM-OP 系统。假设 CPU 有 4 个用户可用的寄存器，而且数据和地址都是 8 位的。假设在一个中断中，8 位的寄存器值和 8 位的返回地址通过按位 XOR 哈希，而且哈希值保存在了 CPU 内部。然后这个寄存器值和返回地址被中断服务程序保存在了内存中。如果这个哈希值被保存在了 CPU 内部，阐述如何检测欺骗、拼接和重放攻击。如果某种攻击不能被检测到，确定原因并给出一个安全机制的建议。你可以假设共有 5 个寄存器分别被标为 0 ~ 4，其中寄存器 4 用来存放返回地址。
- 11.37 HTE 是一个微控制器，执行 SP 内部的一个固件。为了简化，参考数据块的一个哈希树（而不是分配给数据块的序列数的哈希树）。在一个程序可以在 DI-SXM 下访问它的动态数据之前，动态数据块的哈希树必须已经存在。假设 SP 实现了 MESI 缓冲协议，描述一个程序的动态数据块的初始哈希树如何被创建的。还假设还有其他 SP 状态位，因此 OS 可以在必要的时候调用固件，也可以选择使能或禁用 HTE 读周期，其被禁用后导致从内存加载的数据块不能被鉴别。数据内存空间可以在编程时被静态声明或在运行时被分配。
- 11.38 假设每个动态数据块使用隔断的序列号。还假设在 DI-SXM 程序运行时，起始地址和动态数据空间的大小都存在于 SP 中。讨论 / 解释当一个短的序列数字溢出时，哈希树是如何更新的。同样也可以参考练习 11.37。
- 11.39 假设一个基于 SP 的系统对 SXM 程序（代码和数据），非 SXM 程序（代码和数据），保护 SXM 程序数据块的哈希树和系统程序（代码和数据）使用不同的虚拟地址。描述一种 SP 可以用于鉴别每一个不同虚拟地址的正确页表的机制。

参考文献

- Abd-El-Barr Mostafa and El-Rewini Hesham, *Fundamentals of Computer Organization and Architecture*, Wiley, 2005.
- Agner Fog, "Branch prediction in the Pentium family," www.x86.org/articles/branch/branchprediction.htm.
- Altera Quartus II, CPLD, FPGA design tool, <http://www.altera.com/>.
- Anderson John A., *Foundations of Computer Technology*, CRC Press, 1994.
- ATI Xenos GPU (for Xbox 360), www.amd.com.
- Buchanan William J., *Introduction to Security and Network Forensics*, CRC Press, 2011.
- Carpinelli John D., *Computer Systems Organization and Architecture*, Addison Wesley, 2001.
- Christof P., Jan P., and Bart P., *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer, 2010.
- Ciletti Michael D., *Starting Guide to Verilog 2001*, Pearson Prentice Hall, 2004.
- Clements Alan, *Principles of Computer Hardware*, Oxford, 2006.
- Culler David, Singh Jaswinder, and Gupta Anoop, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufman, San Francisco, 1999.
- Easttom William, *Computer Security Fundamentals*, 2nd ed., Pearson, 2011.
- Gendrullis Timo, "Hardware-based cryptanalysis of the GSM A5/1 encryption algorithm," thesis, May 2008.
- Hard drive interfaces, <http://www.harddrivereport.com/>.
- Harris David and Harris Sarah, *Digital Design and Computer Architecture*, Morgan Kaufmann, 2007.
- Harvey A. F., Data Acquisition Division Staff, "DMA Fundamentals on Various PC Platforms," National Instruments.
- Hennessy John and Patterson David, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufman, Waltham, 2012.
- Hwang Kai, *Computer Arithmetic Principles, Architecture, and Design*, Wiley, 1979.
- Intel, "Optimization techniques for integer-blended code," <http://download.intel.com/design/pentiumii/manuals/24281603.pdf>.
- Intel QuickPath, <http://www.intel.com/technology/quickpath/introduction.pdf>.
- Katz R. and Borriello G., *Contemporary Logic Design*, Pearson, 2005.
- King S. T., Tucek J., Cozzie A., Grier C., Jiang W., and Zhou Y., "Designing and implementing malicious hardware," In: *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, April 2008.
- Luebke David and Humphreys Greg, "How GPUs work?" *IEEE Computer*, February 2007, 96–100.
- Mano Morris M. and Kime Charles R., *Logic and Computer Design Fundamentals*, 4th ed., Pearson Prentice Hall, 2008.
- Mano Morris M. and Ciletti Michael D., *Digital Design*, 4th ed., Prentice Hall, 2007.
- Marcovitz Alan B., *Introduction to Logic Design*, McGraw-Hill, 2005.
- Microsoft Keyboard scan code specification, <http://www.microsoft.com/>.
- Northbridge and Southbridge, <http://www.nvidia.com/page/home.html>, <http://www.intel.com/products/chipsets/>, <http://www.amd.com/us/PRODUCTS/>.
- Null Lina and Lobur Julia, *Computer Organization and Architecture*, Jones Bartlett Learning, 2012.
- NVIDIA GeForce GPUs, www.nvidia.com.
- Osadchy M., Pinkas B., Jarrous A., and Moskovich B., "Scifi: a system for secure computation of face identification," In: *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
- Patterson David and Hennessy John, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, San Francisco, 2005.
- Saba A. and Manna N., *Digital Principles and Logic Design*, Jones and Bartlett, 2010.

- Saltzer Jerome H. and Kaashoek M. Frans, *Principles of Computer System Design: An Introduction*, http://ocw.mit.edu/resources/res-6-004-principles-of-computer-systemdesign-an-introduction-spring-2009/online-textbook/protection_open_5_0.pdf.
Samsung hard drives, www.samsung.com.
- Shen John P. and Lipasti Mikko H., *Modern Processor Design*, McGraw-Hill, 2005.
- Smith James E. and Pleszun Andrew R., Implementing precise interrupts in pipelined processors, *IEEE Transactions on Computers*, 1988, 562–573.
- Spansion Flash Memory, <http://www.spansion.com>.
- Stallings William, *Computer Organization and Architecture*, Pearson Education, 2010.
- Stallings William, *Cryptography and Network Security*, Pearson Prentice Hall, 4th ed., 2006.
- Tanenbaum Andrew, *Structure Computer Organization*, Pearson, 2006.
- Universal Host Controller Interface (UHCI) Design Guide, <http://download.intel.com/technology/usb/UHCI11D.pdf>.
- Universal peripheral interface slave microcontroller (UPI-42), www.alldatasheet.com.
- USB (universal serial bus), <http://www.usb.org/home>.
- USB 3.0 specification, <http://www.usb.org/developers/docs/>.
- Vahid Frank, *Digital Design with RTL Design, VHDL, and Verilog*, John Wiley and Sons Publishers, 2011.
- Vray Jogn Shaley, *Interprocess Communications in UNIX*, Prentice Hall, 2003.
- Wakerly J. F., *Digital Design: Principles and Practices*, 4th ed., Prentice Hall, 2006.

索引

索引中的页码为英文原书页码,与书中页边标注的页码一致。

2's complement number (二进制补码数), 3
3DNow instruction set (3DNow 指令集), 20
7400 chip series (7400 芯片系列), 76
7-segment display unit (7 段显示单元), 50

A

Access control list (访问控制列表), 469
Access control matrix (访问控制矩阵), 470
Access control (访问控制), 469
Access point (接入点), 417
Address bus (地址总线), 281
Address strobe (地址选通), 380
Addressing modes (寻址模式), 311
AMD opteron processor (AMD 皓龙处理器), 353, 458
AMD phenom processor (AMD 羿龙处理器), 353
AMD processors (AMD 处理器), 307, 353
AMD quad fx platform (AMD quad fx 平台), 299
Analog-to-digital (模数转换, A/D), 9
Antidependence (反相关), 353
Application programming interface (应用编程接口), 544
Application specific ic (专用集成电路, ASIC), 9, 75, 155
Arbitrator (仲裁器), 399
ARM cortex-A8 (ARM cortex-A8), 307, 357
Array divider (阵列除法器), 139
ASCII codes (ASCII 编码), 2
Assembler directive (汇编伪指令), 318
Asynchronous interrupts (异步中断), 402
Atomic bus access (原子总线访问), 384
Attestation identity key (身份证明密钥), 499
Authdata (授权数据), 500, 501, 503
Availability security property (可用性的安全属性), 463, 504, 536

B

Bandwidth (带宽), 64, 92, 282, 536
Basic input/output system (基本输入/输出系统 BIOS), 384
Bell-lapadula security policy (Bell-lapadula 安全策略), 472
Bi-directional (双向). 见 bus
Biased-exponent (偏置指数), 5, 127
Biba security policy (Biba 安全策略), 472
Binary-coded decimal (二-十进制编码, BCD), 50, 184
Binding data to platform (绑定数据到平台), 498
Bit-parallel design (位并行设计), 96
Bit-serial design (位串行设计), 97
Block carry generate unit (块进位生成单元, BCGU), 105
Block cipher (块密码器), 485
Block replacement (块替换), 442
Bootloader (引导加载程序), 277, 390
Borrow look-ahead subtractor (先行借位 (BLA) 减法器), 108
Borrow propagate subtractor (借位传播减法器, BPS), 108
Branch history table (分支历史表), 390
Branch prediction (分支预测), 382
Bridge (桥), 374
Buffer-overflow attack (缓冲区溢出攻击), 465, 534
Bulk USB data transfer (USB 批量数据传输), 398
Bus (总线), 63
Bus master (总线主控), 400

C

Cache coherency protocol (cache 一致性协议), 435

Cache controller (cache 控制器), 444
Cache hit (cache 命中), 429
Cache line (cache 行), 429
Cache miss (cache 缺失), 429, 445
Capability-list access control (容量列表访问控制), 470
Capacity cache miss (容量缓存缺失), 435
Carry generate unit (进位生成单元, CGU), 101
Checksum (校验和), 494, 515
Chinese wall security policy (中国“长城”安全政策), 473
Cipher (密码器), 485
Cipher MAC (消息认证码), 496
Cipher block chaining (密码块链接, CBC), 487
Ciphertext (密文), 485
Clark-Wilson security model (Clark-Wilson 安全模型), 473
Clock cycle (时钟周期), 155
Clock period (时钟周期), 155
Clock signal (时钟信号), 146
Clock skew (时钟脉冲相位差), 156, 198
Clock-to-output. 见 clock-to-q (时钟至输出)
Clock-to-q (时钟至 q), 156
Cloud computing (云计算), 25
Cluster (集群), 25
Code injection (代码注入), 463
Cold cache miss (cache 冷缺失), 435
Communication interface (通信接口), 448
Comparator logic (比较器逻辑), 138
Complex instruction set computer (复杂指令集计算机, CISC), 225, 315
Computational attack (计算攻击), 475
Confidentiality security policy (保密性安全策略), 463
Configurable CPU (可配置 CPU), 218
Configurable logic block (可配置逻辑块, CLB), 76, 178
Configuration USB descriptor (配置 USB 描述符), 418
Conflict cache miss (冲突缓存缺失), 435
Context switch (上下文转换), 451
Control bus (控制总线), 281
Control memory (控制存储器), 227

Control unit (控制单元), 6, 215
Control USB data transfer (USB 控制数据传输), 418
Coordinate rotation digital computer (坐标旋转数字计算机, CORDIC), 20, 235
Corrupter attack (腐蚀者攻击), 475
Counter mode cipher (计数器模式密码器), 488
Cryptography key stream (加密秘钥流), 485
Cryptography key whitening (加密秘钥洗白), 497
Cryptoprocessor (加密处理器), 484
Cycles per instruction (每条指令的周期数, CPI), 335

D

Data bus (数据总线), 281
Data cache (数据缓存), 427
Data dependence (数据依赖), 353
Data Encryption Standard cipher (数据加密标准 (DES) 密码器), 487
Data-parallel computation (数据并行计算), 22
Data path (数据通路), 6, 215, 271, 282, 305, 374, 435, 464
Data storage through hardware (通过硬件的数据存储), 487
DeMorgan's theorem (德摩根定理), 34
Denormal FP number (非规格化浮点数), 5, 128
Deterministic FSM (确定有限状态机), 174
Device controller (设备控制器), 374
Device controller interface (设备控制器接口 DCI), 9, 374
Device driver routine (设备驱动程序), 353
Device descriptor (USB 设备描述符), 418
Digital rights management (数字权限保护), 484
Digital signal processor (数字信号处理器, DSP), 9
Digital-to-analog (数模转换, D/A), 9
Digitizing analog signal (数字化模拟信号), 2
Discretionary access control (自主访问控制), 470
DMA channel (存储器直接存取通道), 400
DMA transfer table (DMA 传输表), 400
Double data rate sdram (双数据率 (DDR) 同步动态存储器), 294
DRAM refresh cycle (DRAM 刷新周期), 276

Dual principle (对偶原理), 37
 Dynamic energy (动态能量), 231
 Dynamic memory (动态随机存取存储器, DRAM), 275
 Dynamic power consumption (动态功耗), 233

E

Edge triggered flip-flop (边沿触发器), 151
 Efficiency (效率), 223, 279, 298, 308
 Embedded systems (嵌入式系统), 9, 374
 Emitter attack (发射者攻击), 475
 Endpoint USB descriptor (端点 USB 描述符), 417
 Error correcting code SDRAM(错误校正码 (ECC) SDRAM), 510
 Error detection and correction (检错与纠错), 192, 198
 Espresso minimization software (Espresso 最小化软件), 54
 Essential prime implicant (基本素蕴含项, EPI), 46
 Exceptions (异常), 401
 External cache hit (外部 cache 命中), 445

F

Fair memory access scheduler (公平内存访问调度程序), 536
 False-sharing cache miss (错误共享 cache 缺失), 444
 Fault tolerant FSM (容错 FSM), 174
 Feature size (特征尺寸), 1
 Field programmable gate array (现场可编程门阵列, FPGA), 9, 155
 Fifo buffer (先入先出缓冲器), 185
 Firmware (固件), 351, 484
 Flame virus (火焰病毒), 471
 Flash memory (flash 存储器), 274, 390
 Floating-point number (浮点 (FP) 数), 5, 126
 Floating point operations per second (每秒浮点运算次数, FLOPS), 24, 224
 Floating point unit (浮点单元, FPU), 98
 Forward branching (转发分支), 344
 Forwarding unit (转发单元), 330
 Frame (框架), 398, 416
 Front-side bus (前端总线, FSB), 376

Fully associative mapping cache (全相联映射 cache), 433, 456
 Fused operation (混合操作), 217

G

Glitch (故障), 60, 147
 Global branch predictor (全局分支预测), 352
 Graphic processing unit (图形处理器, GPU), 9, 20, 269
 Gray code (格雷码), 184

H

Hamming code (汉明码), 192, 510
 Hamming distance (汉明间距), 192
 Hardware backdoor (硬件后门), 473
 Hardware description language (硬件描述语言, HDL), 2, 16
 Hardware interrupts (硬件中断), 401
 Hardware trojan (硬件木马), 473
 Hash value (哈希值), 494
 Hashed MAC (基于哈希的 MAC), 497
 Hazard (冒险)。见 Glitch
 Hazard unit (冒险单元), 332
 Heterogeneous cores (异构核), 22
 High impedance (高阻抗), 61, 284
 Hit ratio (命中率), 431
 Homomorphic computation (同态计算), 520
 Host controller interface (主控制器接口), 9, 374
 Hot-spot (热点), 449
 Hybrid FSM (混合 FSM), 172, 185
 Hypertransport interconnect (超传输互联), 378

I

I/O controller hub (I/O 控制器集线器), 377
 I/O ports (I/O 端口), 9, 374
 Implicant (蕴含项), 46
 Implicit latch (隐式锁存器), 158, 179
 Information flow tacking (信息流定位), 464, 535
 Input port (输入端口), 387
 Instruction cache (指令 cache), 427
 Instruction cycle (指令周期), 310
 Instruction pipeline (指令流水线), 307

Instructions per cycle (每周指令数, IPC), 308, 340
Integrated chip (集成电路芯片), 1
Integrity security property (完整性的安全属性), 463
Intel core i7 (英特尔酷睿 i7 处理器), 21, 23, 308, 358
Intel itanium processor (英特尔安腾处理器), 21, 356
Intel pentium IV processor (英特尔奔腾 IV 处理器), 360
Intel xeon processor (英特尔至强处理器), 430, 442
Interface USB engine (USB 接口引擎), 423
Integer unit (整数单元), 98
Interleaving (交叉), 295, 300, 341
Interrupt acknowledge (中断确认), 407
Interrupt-driven I/O (中断驱动输入/输出), 393
Interrupt handler (中断处理程序), 393, 522
Interrupt priority (中断优先级), 393
Interrupt request (中断请求), 395
Interrupt structure (中断结构), 393
Interrupt USB data transfer (USB 中断数据传输), 398, 420
Interrupt vector table (中断向量表), 405
Invalidation cache protocol (失效缓存协议), 444, 446
Isochronous USB data transfer (USB 准同步数据传输), 398, 417

J

JK flip-flop (JK 触发器), 157

K

K-Map minimization rules (K-Map 最小化规则), 46
Keyboard key matrix (键盘按键矩阵), 391
Keyed-hash (基于关键字的哈希), 496, 500, 510

L

Latency (延迟), 299, 375, 378, 385
Leakage current (漏电电流), 234, 276
Leaking information (信息泄露), 476, 513, 516,

522, 535

Level 1 cache (一级 cache), 428
Level 2 cache (二级 cache), 428
Level 3 cache (三级 cache), 430, 442
Line memory (行存储器), 435
Linear feedback shift register (线性反馈移位寄存器, LFSR), 485
Local memory (本地存储器), 299
Local branch predictor (本地分支预测), 350, 392
Logic gates (逻辑门), 10
Logic product term (逻辑乘积项), 34
Logic sum term (逻辑和项), 36

M

Machine instruction (机器指令), 8, 311
Mandatory access control (强制访问控制), 470, 473, 484
Mealy FSM, 172
Memory access time (存储器存取时间), 289
Memory authentication (内存认证), 551
Memory banks (存储片), 279
Memory cell (存储单元), 274
Memory controller hub (存储器控制单元, MCH), 374
Memory cycle (存储周期), 289
Memory management unit (存储器管理单元, MMU), 452
Memory-mapped I/O (内存映射 I/O), 386
Memory row activation (存储器行激活), 277
Message digest (消息摘要), 496
Message passing system (消息传递系统), 24
Metastability (亚稳态), 152
Micro-operation (微操作), 227
Microcontroller (微控制器), 374, 390
Microinstruction (微指令), 227
Microprogram (微程序), 227
Microprogrammed control (微程序控制), 225
Million instruction per second (每秒百万条指令, MIPS), 224
MIPS processor (MIPS 处理器), 225, 307, 316, 336
Miss ratio (缺失率), 431

Mnemonic opcode (助记码), 310
 Moore FSM, 172
 Moore's law (摩尔定律), 1
 Motherboard (主板), 377
 Multi-lateral security policy (多边安全政策), 472
 Multi-level security policy (多级安全策略), 472, 503
 Multiple instruction multiple data (多指令多数据, MIMD), 22
 Multiprogramming (多道程序设计), 401, 450
 Multithreaded programming (多线程编程), 22, 308, 362
 Multithreading (多线程), 22, 341

N

National institute of standards and technology (国家标准技术局, NIST), 487, 498
 Net-list (网表), 17, 75
 Network adaptor (网络适配器), 386
 Non-computational attack (非计算性攻击), 475
 Non-return-to-zero inverted (反向不归零, NRZI), 212, 270, 415
 Non-uniform memory access (不统一存储器存取, NUMA), 378, 448
 Non-volatile memory (非易失性存储器), 274
 Nonce (临时值), 499
 Normal FP number (规格化浮点数), 5, 128
 Normalizing FP result (规格化浮点数结果), 133
 Northbridge (北桥), 377

O

Object code (目标代码), 310
 Out-of-order execution (乱序执行), 357
 Output dependence (输出相关), 353
 Output port (输出接口), 387

P

Packet (包), 283, 415
 Page fault (缺页), 401, 430
 Page mode access (页面模式访问), 279
 Paging (分页), 430
 Parity bit (校验位), 195
 Parity generator (奇偶发生器), 198

Pass transistor (晶体管), 275
 Physical attacks (物理攻击), 25
 Physical memory (物理存储器), 429
 Physically addressed cache (物理寻址 cache), 458, 521
 Pipeline chart (流水线图表), 220
 Pipeline flush (流水线清空), 331
 Pipeline stage (流水线阶段), 220
 Placement-and-route (布局和布线), 79
 Plug and play devices (即插即用设备), 373
 Point-to-point communication (点对点通信), 64, 378
 Port-mapped I/O (端口映射 I/O), 346
 Precise interruption (确定中断), 361
 Predicated instruction (预测指令), 356, 535
 Prime implicant (素蕴含项), 46
 Primitive gates (基本门), 80
 Priority encoder (优先编码器), 73
 Private key (秘钥), 489
 Process switch (进程切换), 452
 Process (进程), 451
 Processing core (处理核心), 2
 Program counter (程序计数器)。见 Program pointer
 Program pointer (程序指针), 318
 Programmable logic device (可编程逻辑器件, PLD), 75
 Programmed I/O (程序控制 I/O), 393
 Propagate-generate unit (传播生成单元, PGU), 101
 Pseudo instruction (伪指令), 318
 Public key (公开秘钥), 489
 Public key infrastructure (公钥密码, PKI), 491

Q

Quickpath interconnect (快速通道互联), 378
 Quine-McCluskey algorithm (奎因-麦克拉斯基算法), 51

R

Random access memory (随机存取存储器, RAM), 274
 Randomized encryption (随机加密), 516
 Read after write hazard (写后读 (RAW) 冒险), 353

Reciprocal division algorithm (例数除法算法), 139
Redundant array of independent disks (独立磁盘
冗余阵列, RAID), 385
Register renaming (寄存器重命名), 357
Register transfer language (寄存器传送语言, RTL),
16
Register window (寄存器窗口), 309
Remote memory (远程存储器), 299
Replay attack (重放攻击), 481, 504, 514, 519
Restoring division algorithm (恢复除法算法), 124
Reverse polish notation (逆波兰表示法), 313
Ripple carry adder (行波进位加法器, RCA), 99
Rotations per minute (每分转速), 385
Rounding error (舍入错误), 133, 217

S

Sampling rate (采样率), 2
Scan code (扫描码), 392
Score boarding (记分牌), 358
Secret root key (加密根键), 499
Secure execution environment (安全执行环境),
484, 509
Secure execution mode (安全执行模式, SXM), 484,
509
Secure root hash (安全根哈希), 504
Security key storage (安全密钥存储), 499
Seek time (查找时间), 385
Sense amplifier (放大器), 277
Sensitivity list (敏感信号表), 86, 158
Server overload (服务器负载超重), 464
Session key (会话密钥), 512
Shared cache (共享 cache), 430
Shared memory system (共享存储系统), 22
Shoot-through current (贯通电流), 232
Side channel attacks (侧信道攻击), 474
Sign extension (符号扩展), 4
Signal chasing (信号追逐), 148
Signal fall time (信号下降时间), 58, 234
Signal handshaking (信号握手), 381
Signal polarity (信号极性), 31
Signal rise time (信号上升时间), 58, 234
Signature security key (有签名的安全密钥), 499
Signed magnitude number (带符号数), 3, 127

Silicon graphics' sgi Altix 4700 system (硅图公司
的 SGI Altix 4700 系统), 299
Single instruction multiple data (单指令多数据,
SIMD), 20, 98, 218, 308
Single instruction single data (单指令单数据, SISD),
22
Snoop controller (监听控制器), 444
Software interrupt (软件中断), 401
Southbridge (南桥), 377
Sparc processor (Sparc 处理器), 307, 344, 345
Spatial locality (空间局部性), 431
SPEC CPU2006 (CPU2006 规范), 224
SPEC89 (SPEC89 规范), 352
Speculative execution (预测执行), 356, 529
SPECviewperf (SPECviewperf 规范), 224
Speedup (加速比), 223
Splicing attack (拼接攻击), 481, 509
Split transaction (分离事务), 383
Spoofing attack (欺骗攻击), 481, 509, 534
Static memory (静态存储器), 275
Static power consumption (静态功耗), 234
Status change USB endpoint(状态更改 USB 端点),
417
Steaming SIMD extension (流式 SIMD 扩展, SEE),
20
Stream cipher (流密码器), 485, 486
Stuxnet malware (恶意软件), 472
Superpipelining (超流水), 340
Superscalar processor (超标量处理器), 340
Synchronizing flip-flop (同步触发器), 203
Synchronous interrupts (同步中断), 401
System-on-chip (片上系统, SoC), 9, 378

T

T flip-flop (T 触发器), 157
Tag memory (标记存储器), 435
Temporal locality (时间局部性), 431
Test-bench (测试台), 79
Thermal design power (热设计功率), 23, 235
Third-party modules (第三方模块), 464
Thread (线程), 22, 361
Thread-level parallelism (线程级并行, TLP), 363
Thread switch (线程切换), 452

Threat vector (威胁向量), 509
Throughput (吞吐量), 24, 223, 307
Time slice (时间片), 402
Timing attack (定时攻击), 474, 489
Transceiver (收发器). 见 bus
Transient fault (瞬时故障), 166
Transistor CMOS (晶体管, CMOS), 12
Trap (陷阱), 401
True color mode (真彩色模式), 2
True-sharing cache miss (真实共享 cache 缺失), 444
Trusted computing base (可信计算基, TCB), 465, 508
Trusted firmware module (可信固件模块), 484
Trusted hardware module (可信硬件模块), 484
Trusted platform module (可信平台模块), 484
Trusted software module (可信软件模块), 484, 536, 551

U

Unicode (统一字符编码标准), 2

Uniform memory access (统一存储器访问, UMA), 378, 430
Universal serial bus (通用串行总线, USB), 9, 374
Update cache protocol (更新 cache 协议), 442

V

Vertex transformation (顶点转换), 20, 261
Virtual memory (虚拟存储器), 319, 429
Virtually addressed cache (虚拟地址寻址 cache), 452
Volatile memory (易失性存储器), 274
Von Neumann machine (冯·诺依曼计算机), 7

W

Wait cycle (等待周期), 380
Wait queue (等待队列), 401
Wait state (等待状态), 380
Warehouse computing (仓储计算), 25
Wired-logic (线逻辑), 62
Write after read hazard (写后读 (WAR) 冒险), 353
Write after write hazard (写后写 (WAW) 冒险), 353

数字逻辑设计与计算机组成

Digital Logic Design and Computer Organization with Computer Architecture for Security

计算机系统硬件课程对于教师和学生都是一大挑战，本书是迎难而上的一部力作，展现了作者对硬件的精妙理解，既涵盖数字逻辑设计与计算机组成这些传统内容，又创新式地引入了安全体系结构问题。本书的深度高于大部分同类教材，但同时深度与广度更为平衡，循序渐进地铺就了从基础电路到计算机系统的硬件/软件贯通之路。

本书特色

- 面向教学的内容组织模式。从组合电路、时序电路的简单设计逐步进阶复杂设计，包含丰富的实例和练习，并配有Verilog代码，同时鼓励学生使用各类工具来提高设计效率。
- 关注系统视角的安全问题。用完整的一章深入探讨计算机系统如何利用硬件来支持安全的体系结构，包含访问控制、硬件安全策略机制以及加密技术等诸多概念和方法。
- 突出现代计算机设计理念。移动计算和高性能计算对计算机基础结构的影响日益深远，本书对这一趋势的关注和解读，对于学生学习后续课程和从事相关工作都大有助益。

作者简介

尼克罗斯·法拉菲（Nikrouz Faroughi）计算机科学和工程领域教授，拥有密歇根州立大学电子工程博士学位，现任职于加利福尼亚州立大学萨克拉门托分校，曾在Intel公司担任顾问和技术经理。

全球智慧中文化

<http://www.mheducation.com>



投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机\计算机组成

ISBN 978-7-111-57061-5



9 787111 570615 >

定价: 89.00元